Part II  Function-Based Computing

Chapter 4  Lambda Calculus
4.1 Syntax of Lambda Calculus 67
4.2 General Model of Computation 68
4.3 Standard Simplifications 69
4.4 Identifiers 71
4.5 Substitution Rules 72
4.6 Conversion Rules, Reduction, and Normal Order 74
4.7 The Church-Rosser Theorem 77
4.8 Order of Evaluation 80
4.9 Multiple Arguments, Currying, and Naming Functions 81
4.10 Basic Arithmetic in Lambda Calculus 83
4.11 Boolean Operations in Lambda Calculus 86
4.12 Recursion in Lambda Calculus 88
4.13 Problems 90

Chapter 5  A Formal Basis for Abstract Programming 94
5.1 A Basic Syntax 95
5.2 Constants 96
5.3 Function Applications 97
5.4 Conditional Expressions 98
5.5 Let Expressions—Local Definitions 99
5.6 Recursive Definitions 103
5.7 Global Definitions 106
5.8 Higher-Order Functions 106
5.9 An Example—Symbolic Differentiation 108
5.10 Problems 111

Chapter 6  Self-Interpretation 113
6.1 Abstract Interpreters 115
6.2 Lambda Expressions as S-Expressions 119
6.3 An Expanded Interpreter 124
6.4 Association Lists 127
6.5 Closures 133
6.6 Recursive Closures 135
6.7 Closing the Loop—Read-Eval-Print 137
6.8 Problems 138

Chapter 7  The SECD Abstract Machine 140
7.1 List Memory 142
7.2 Basic Data Structures 143
7.3 The SECD Machine Structures 150

Chapter 8  Memory Management for S-expressions 179
8.1 Allocation 180
8.2 Mark-Sweep Collection 183
8.3 Reference Counts 192
8.4 Garbage Compaction 195
8.5 Alternative List Representations 204
8.6 Problems 211

Chapter 9  Demand-Drive Evaluation 213
9.1 Explicit Delays and Forces 215
9.2 Program Examples 217
9.3 Recipes, Promises, Futures 219
9.4 Lazy Evaluation 224
9.5 Streams 229
9.6 Continuations 233
9.7 Problems 241

Chapter 10  LISP: Variations and Implementations 243
10.1 The Original LISP 244
10.2 Scheme—A Pure LISP 257
10.3 Common LISP—A Modern Standard 265
10.4 MultiLISP—A Parallel LISP 270
10.5 The CDR Machine 273
10.6 CDR Derivatives 278
10.7 Other LISP Machines 285
10.8 Benchmarking 291
10.9 Problems 295

Chapter 11  Combinators and Graph Reduction 296
11.1 Basic Combinators 297
11.2 Bracket Abstraction 300
11.3 Additional Combinators 303
11.4 Optimizations 309
11.5 Graph Reduction 312
11.6 Supercombinators 322
11.7 Combinator Machines 326
11.8 The G-Machine 332
11.9 Problems 336
CHAPTER 11

COMBINATORS AND GRAPH REDUCTION

As described here, lambda calculus is a complete theory that describes much of mathematics with a single simple semantic model. Its only major difficulties are that care must be taken with identifier names (the renaming rule) and that a relatively large number of different functions is needed for even simple expressions.

Combinator theory is an interesting variation of lambda calculus that has neither of these difficulties. It consists of a minimal set of functions from which equivalents to all lambda expressions can be constructed by a very minimal translation algorithm. Further, the resulting expressions are simple compositions of combinators, meaning that we need no other lambda functions or identifiers (bound or otherwise).

The origins of this theory come from the observation that all identifiers in a lambda expression really "steer" copies of actual arguments to appropriate spots in a function's body. If we permit multiple argument functions, then it is possible to conceive of doing the same thing by creating functions which simply reorder their arguments in ways that end up duplicating the effects of this steering. These reordering functions are combinators, and they can be fully described without need of identifiers or the substitution rule.

The following subsections discuss combinator theory in more detail. First will come a description of a minimal set, followed by the process by which arbitrary lambda expressions can be converted to combinator form. This involves the bracket abstraction process, by which the identifiers are "abstracted" out of an arbitrary lambda function. Following this will come a discussion of other useful combinators beyond the basic set, and optimization algorithms that can take one combinator expression and simplify it by using these additional combinators.

Next will come a model of computation that represents combinator-based expressions as graphs, and performs computations on them by graph reduction of subtrees. Such reduction strategies seem to violate the long-held principle of avoiding self-modifying code, but they have the advantages of extreme simplicity and near-automatic lazy evaluation.

Yet another optimization technique of growing importance spends effort at compile time to develop supercombinators, which are highly optimized for particular subexpressions in the program being compiled. This approach forms the basic of much of modern research into implementing functional languages using graph reduction. Section 11.6 describes the process of identifying such supercombinators and how to optimize their construction and use.

The final sections describe several interesting machines that rely on combinator theory for their basic operation. In particular, the G-Machine represents yet another abstract machine of growing importance in the compilation and execution of functional programs.

Perhaps the key ideas a reader should take away from this chapter are the concept of a combinator as an alternative to substitution, and graph reduction as a different way of structuring and managing a functional evaluation. For alternative references, the reader is referred to Curry and Feys (1958), Seldin and Hindley (1980), Turner (1979a), Burge (1975), Jones and Muchnick (1982), and Diller (1988).

11.1 BASIC COMBINATORS

One of our first notational simplifications for lambda calculus was to give names to standard lambda functions that are used over and over in real expressions (such as integers, "+-", "*", "/", "..."). Whenever such labels show up in an expression they always mean the equivalent function. Since their meaning is "constant," it is not unreasonable to formally call such functions constants.

When confronted with such an obviously useful notation, one of the first reactions of a good mathematician is to see what is the computational power of a sublanguage consisting only of such constants, and whether one set of constants is more useful than another. This was done with lambda calculus, with amazing results. All of lambda calculus can be developed from a set of only three constants (one of which is actually redundant). Further, it can be done with no lambda function definitions, just applications of one constant to another.

11.1.1 Constant Applicative Forms

The particular sublanguage of interest is called the language of constant applicative forms, with a single syntactic class called a c.f. The basic sy-
tax for caf is the essence of simplicity: A caf is either a constant or the application of two caf:

\[(\text{caf}) := (\text{constant}) \downarrow ((\text{caf}) \downarrow (\text{caf}))\]

Not present in this definition are free variables and lambda functions (\(\lambda x . . .\)).

As with lambda calculus, when multiple arguments to a function are involved, we will delete the innermost "(\text{),}" permitting caf's of the form:

\[(\text{caf}) := (\text{constant}) \downarrow ((\text{caf}) \downarrow (\text{caf}))^*\]

Also, as with lambda calculus applications, the leftmost caf in an application is the function, those to its right are its actual arguments. And the fundamental meaning of such multiple-argument forms is the same as for lambda calculus. The innermost function is curried by accepting one argument at a time and producing a function of one fewer arguments.

This is lambda calculus without the lambda function definition (and as such looks a lot like s-expressions). We cannot define new functions as lambda expressions; we can only build functions from whatever constant functions are provided.

The constants for this sublanguage will be chosen carefully. What we want are functions that, when given caf's as arguments, always give caf's as a result. No free variables or embedded lambda functions that are not constants will appear. Further, we want to be able to define their operation without concern about substitutions and possible identifier clashes. In fact, there will be no need even to introduce the idea of an identifier, binding variable, or the like, in the semantics.

There are two categories of such constants. The first is the common set of arithmetic functions, integers, booleans, etc., which we know full well how to translate to pure lambda expressions, but would rather not. These are to be built-in to our language and may vary from one language to another. Functions in this class have the property that both their domains and their ranges are created from objects which we also call constants. For example, "+" maps from pairs of integers to integers (or in curried form from integers into functions of integers to integers). The result is that a caf created from such constants will, when evaluated, return another expression which is still a caf.

11.1.2 S, K, and I

The second set of constants, called combinators, are more general and, in a sense, more basic. These constants represent some particularly well-behaved functions out of which all others can be built, including the first set of constants. By definition, a combinator is any lambda function which:

- Has no free variables in it (no identifiers not surrounded by a controlling \((\ldots)\))
- Has a body in a caf form (or convertible to caf form)
- Or, more simply, when given arbitrary caf's as arguments, always returns a caf result.

To make a combinator a constant, we will give each one a unique label and use that in place of the combinator's full lambda equivalent.

Although there are an infinite number of such combinators, some are more important than others. In particular, the three combinators S, K, and I defined in Figure 11-1 are usually considered most key. If A, B, and C are arbitrary caf's, then applying S, K, or I to them results in new caf's that are just rearrangements of the input. Computing such applications are thus much simpler than for more general lambda functions; we need no internal detail on identifiers or on the possible complexities resulting from substitutions and renamings.

In contrast, consider the function \((\lambda x(\lambda y(\lambda y x) y))\). Applying this to an arbitrary expression A cannot be carried out and written down without knowing what free variables exist in A, and we still end up with an unremovable lambda expression in the middle of the result. In this form it is definitely not a candidate for inclusion as a basic combinator constant.

Each of the combinators in Figure 11-1 has a relatively easily understood interpretation. The I combinator is basically the identity function, which returns its single argument totally unmodified. The K combinator is a function that takes two arguments and returns a copy of the first. The second is discarded. In its curried form, it takes an arbitrary object A and creates from it a function which takes any other object as an argument and always returns A. For this reason it is known as the constant combinator or elemental cancellator. It is equivalent to our previous definition of T.

The S combinator is a function that takes three arguments A₁, A₂,
and \( A_3 \), and returns an application where \( A_1 \) and \( A_2 \) are both treated as functions, and are given as arguments a copy of \( A_3 \). The result of applying \( A_1 \) to \( A_2 \) is then used as a function over \((A_3 \, A_3)\). Since \( S \) distributes its third argument over two other applications, it is known as the \textit{distributed application combinator} or sometimes as the \textit{elemental combinator}.

The typical use of the \( S \) combinator is in its curried form, when it is given two functions and creates as a result a new function that accepts a single argument into an application involving the first two. As an example of this, consider the function \( S \, K \, K \). The lambda equivalent is \((\lambda z (K \, (K \, z)))\). Using the definition of \( K \), the body of this is the first argument \( z \), that is, \((\lambda z z)\). Thus \( S \, K \, K \) is the same as the function \( I \).

Other simple examples include the \textit{caf} \((S \times I)\), which is equivalent to a function which squares its argument, or \((S + \sqrt{t})\) which is equivalent to the function \((\lambda x x + (\sqrt{x} \cdot x))\).

The next section will expand upon the generality of this set of combinators.

### 11.2 Bracket Abstraction
(Schonfinkel, 1925; Turner, 1979b; Hughes, 1982)

In terms of computation, the most relevant piece of lambda calculus syntax is the application, where the function is in prefix form to the left of its argument. Conceptually, this is simple to evaluate; we give the function its argument value, and replace the pair by the result.

The actual complexity (such as it is) comes into play when we look at the definition of a typical function in the form \((\lambda x \, M)\). The \( x \) is a placeholder for the actual argument, and all free occurrences of \( x \) will receive copies of that argument when the application is reduced. Further, there are no constraints on how many and where the free \( x \)'s can be in \( M \). They can be intermixed freely with other arbitrary lambda expressions. The result is that intermediate steps in the function's evaluation must constantly involve taking strings of characters, looking them over for certain characters in special situations, dismantling the strings at these points, substituting new strings, and reassembling.

Now consider what would happen if we could always rewrite the arbitrary body expression \( M \) of a function \((\lambda x \, M)\) in a form where there is at most one free \( x \), it is always the last character, and the rest of the expression is built out of well-behaved constants such as combinators. In other words, we find some caf \( N \) with no free \( x \)'s such that \((\lambda x \, M)\) and \((\lambda x \, N \, x)\) are the same function, i.e.,

\[
\text{for all } A, (\lambda x \, M) A = (\lambda x \, N \, x) A = N A \]

The existence of such an \( N \) has some profound consequences. Using the eta conversion rule discussed in Chapter 4, the entire expression \((\lambda x \, M)\) can be replaced by simply \( N \). We have "abstracted out" the need for the identifier \( x \), and "compiled" the original function into a string of simple constants without identifiers or "\( \lambda \)'s."

The processing of abstracting \( x \) out of \( M \) is called \textit{bracket abstraction} and is written \([x]M\). This name and notation come from the analogy that we are "reversing" the basic substitution mechanism of applications; we are pulling something out of an expression rather that putting something in.

The interested reader is also referred to the work by Berkling (1986) called \textit{head order reduction}, where something like bracket abstraction in reverse is used to convert a lambda expression to one where all the functions have the form \((\lambda x_1 \ldots x_n x)\). These functions are called \textit{selectors}, and they can be implemented easily as indices into the environment.

### 11.2.1 Basic Translation

Figure 11-2 lists the major syntax rules for forming a lambda calculus expression \( M \), and for each one shows how it would be converted into an equivalent caf form \([x]M\) when some variable \( x \) is to be abstracted out. Thus, in all cases, we are free to replace an expression \( M \) wherever it appears by the form \((\lambda x \, M \, x)\).

The cases covered by the second rule of this figure are worth closer consideration. This rule governs the case when the expression \( M \) is a single symbol that is not the same as the identifier being abstracted out. Thus \( M \) could be any other variable \( x \), or any constant, combinator, built-in function, integer, etc. Thus \([x] x \Rightarrow (K \, I)\), just as \([x] x \Rightarrow (K \, S)\).

To repeat what this table means, each entry in the \([x]M\) column is an expression which, when applied as a function to the variable being abstracted \( x \) in this case, will yield an expression which is totally equivalent to the original \( M \). This permits us to take lambda functions of the form \((\lambda x \, M)\) and replace them by \((\lambda x \, [x]M)\) or, even more simply, by \([x]M\).

The proofs that each \([x]M\) has the property that \((\lambda x \, M) \Rightarrow M\) are direct. For example, \((\lambda x \, x) x \Rightarrow x \) by the definition of \( I \). Also, \((\lambda x \, y) x \Rightarrow (\lambda y) y \Rightarrow y \) by the definition of \( K \). Again note that this holds for constants as well as identifiers other than \( x \).

<table>
<thead>
<tr>
<th>Syntax of ( M )</th>
<th>Example</th>
<th>Equivalent ([x]M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;id&gt; )</td>
<td>( x )</td>
<td>( I \ ) the identifier ( x )</td>
</tr>
<tr>
<td>( &lt;id&gt; )</td>
<td>( y )</td>
<td>( (K , y) ) where ( y ) a constant or identifier ( \neq x )</td>
</tr>
<tr>
<td>( &lt;application&gt; )</td>
<td>((F , A))</td>
<td>((S , [x]F , [x]A))</td>
</tr>
<tr>
<td>( &lt;function&gt; )</td>
<td>((\lambda x , E))</td>
<td>((K , ([x]E))) Binding variable is ( x )</td>
</tr>
<tr>
<td>( &lt;function&gt; )</td>
<td>((\lambda y , E))</td>
<td>((x , ([y]E))) where ( y \neq x )</td>
</tr>
</tbody>
</table>

**FIGURE 11-2**
Bracket abstraction process.
Second, to show that \((\lambda x)(PQ)) = \langle S \ [x]P \ [x]Q \rangle\), we apply \(x\) to both sides and reduce:

\[
(\lambda x)(PQ)x = \langle S \ [x]P \ [x]Q \rangle x \Rightarrow [x]P x ([x]Q x)
\]

which is equivalent to \(P Q\). (We assume recursively that the process worked correctly for \([x]P\) and \([x]Q\); that is, \([x]P = P\) and \([x]Q = Q\).)

The final two rules of Figure 11-2 handle the case where we are attempting to abstract an identifier out of a lambda function. We first abstract the function's identifier out of its body, and then abstract the first identifier out of the result. As in the general case, if the binding variable is the same as \(x\), then the process reduces to one of these rules. The second case, however, is always valid.

Proof of this last rule is also recursive. If the bracket abstraction process works, then \((\lambda y)(E) = ([y]E)\), which is an expression without lambda functions. The expression may, however, have embedded \(x\)'s in it. Thus, abstracting the \(x\) from it yields a caf.

These final rules also indicate how to get the abstraction process started in the first place. If our goal is to translate a lambda function to its equivalent caf form, and we start by abstracting out of the function's body the function's first binding variable. To convert a multiple argument function such as \((\lambda x_1 \ldots x_n)(M)\) to combinator form, we abstract out the first identifier \(x_1\), which in turn requires abstracting out \(x_2\) first. This requires abstracting out \(x_3\), and so on until all the argument identifiers are abstracted out, as follows:

\[
(\lambda x_1 \ldots x_n)(M) = (\lambda x_1(\lambda x_2(\ldots(\lambda x_n)(M)\ldots))
\]

11.2.2 A Simple Compiler

To complete our picture of this whole compilation process, Figure 11-3 gives an abstract program that accepts an arbitrary lambda function and converts it into a pure caf. The special case for function definitions having the same binding variable as \(x\) is not handled here—the general case is applicable, albeit with more work. Abstract syntax predicates and functions are used to take the input apart, test it, and put the result back together. Their meaning should be obvious.

The process of creating a compiler for s-expressions is direct replacement of the abstract functions by the appropriate list processing functions, and is left as a problem for the reader.

Figure 11-4 gives several simple examples of this translation process plus a check for each one that the resulting caf is correct.

compiler(e) = "compiles lambda definition \(e\) into combinators" =
abstract(get-id(e),get-body(e))
abstract(x,e) = "abstract out identifier \(x\) from \(e\)" =
if is-a-constant(e)
  then create-constant(e)
else create-application(K,e)
else if is-a-function(e)
  then abstract(x,compile(e))
else "\(e\) is an application!"
create-application
  create-application(S, abstract(x,get-function(e)),
                      abstract(x,get-argument(e)))

FIGURE 11-3
Lambda to combinator abstract compiler.

11.2.3 A Larger Translation

For larger problems the process is the same, only more tedious, particularly when we encounter an expression of the form \((E_1E_2 \ldots E_n)\) [remember that this is equivalent to \(\langle \ldots \langle E_1E_2 \rangle \rangle \ldots \rangle\)].

The third rule of Figure 11-2 gets used over and over again on essentially the same expressions. Each time it is applied, another \(S\) combinator is added, and bracket abstraction of one more \(E_i\) is initiated.

The process is so common and so mechanical that it is worth working out once and using thereafter as a standard expansion. Figure 11-5 gives this rule and its proof.

As a larger example, Figure 11-6 gives a detailed example of the conversion of the successor function \((\lambda x y z)(xyz))\) to a caf form using only \(S\), \(K\), and \(I\) as constants.

Finally, as an example of a bigger operation, Figure 11-7 takes the main body of the factorial function and converts it to combinators, although in this case we do assume built-in functions for basic math, integers, and the conditional if. A later section will complete the conversion of the full factorial to combinator form, including the necessary recursion control.

11.3 ADDITIONAL COMBINATORS

(Turner, 1979b; Buge, 1975; pp.41-49)

Two things are obvious from Figure 11-6. First, the compilation process seems to explode all out of proportion to the complexity of the original function. In fact, expressions of \(n\) symbols often balloon up something approaching \(2^n\) symbols when put in basic combinator form. Second, in such expansions there are many repetitions of the same subexpressions [e.g., \((S(KK)(KS))\)]. Both need to be reduced to make combinator theory a practical basis for practical computers.
Example: \((\lambda x \cdot 1 \cdot x)\) where + and 1 are built-in constants
\( \Rightarrow [x] + 1 x \)
\( \Rightarrow S(x)(x + 1) 1 \)
\( \Rightarrow S (S (K \cdot 1) \cdot ((x) x)) 1 \)
\( \Rightarrow S (S (K \cdot 1) \cdot 1) 1 \)

Check: evaluate \((S (K \cdot 1) \cdot 1) 1 3 \)
\( \Rightarrow (S (K \cdot 1) \cdot 1) 3 (1 3) \)
\( \Rightarrow (K + 1) 3 (1 3) \)
\( \Rightarrow + (K 1) 3 (1 3) \)
\( \Rightarrow + (1 3) = + 1 3 = 4 \)

(a) A simple function with constants.

Example: \(true = (\lambda x \cdot (\lambda y \cdot x))\)
Note: this is also the definition of \(\text{K}\).
\( \Rightarrow [x] (\lambda y) x \Rightarrow [x] (K x) \Rightarrow S (K K) (\lambda x) \Rightarrow S (K K) 1 \)

Check: Evaluate \((S (K K) \cdot 1 \cdot A) B \)
\( \Rightarrow (K K) A (1 A) B \)
\( \Rightarrow (K A) B \Rightarrow (I A) \Rightarrow A \)

(b) A nested function.

Example: \((\lambda x 1 \cdot (x \cdot x x) x)\)
Note: if fully parenthesized: \((\lambda x 1 \cdot ((x \cdot x x) x))\)
\( \Rightarrow (S (S (x + 1) (x x))) 1 \)
\( \Rightarrow (S (S (K + 1) (S (x x) (x x))) 1 \)
\( \Rightarrow (S (S (K + 1) (S (S (K x) (x x))))) 1 \)
\( \Rightarrow (S (S (K + 1) (S (S (K x) (I 1))))) 1 \)

Check: Evaluate \((S (S (K + 1) (S (S (K x) (I 1)))) \cdot 1) 3 \)
\( \Rightarrow (S (K + 1) (S (S (K x) (I 1)))) 3 (1 3) \)
\( \Rightarrow (K + 1) 3 (S (S (K x) (I 1))) (1 3) \)
\( \Rightarrow + (S (K x) (I 1)) 3 (1 3) (1 3) \)
\( \Rightarrow + (K x) 3 (1 3) (1 3) \)
\( \Rightarrow + (1 3) (1 3) (1 3) \)
\( \Rightarrow + (3 3) \Rightarrow 27 \)

(c) Nested abstractions.

FIGURE 11.4
Some simple combinator compilations.

cursion in lambda expressions. Such techniques are discussed more fully in the next section.

Figure 11-8 gives a table of useful combinators. As before, each one simply rearranges its arguments, guaranteeing that if the arguments are caf's, then the result is a caf. Also note that many of these combinators have shown up before as equivalents for well-known objects such as integers.

It is worth the reader's time to prove that the operation of each combinator (its semantics) matches the lambda definition. It is also instructive to convert each of these lambda definitions into strings of pure \(S\), \(K\), and \(I\) combinators. This both reinforces the compilation process and indicates the power of these new combinators.

Several combinators from this set are worth discussing further. First, \(B\) (the basic composition combinator) accepts three arguments—the first two of which are treated as functions. The result is the first applied to the result of applying the second to the third. This is exactly composition by previous definitions.

Next, \(C\), the reversal combinator, takes three arguments and returns three, but with the last two reversed. \(W\), the repeat combinator, accepts two arguments and returns three, where the last is the second repeated.

The primitive recursion combinator \(R\) implements the equivalent of a do-loop in conventional languages. It accepts three arguments: a termination value, a function, and an index. The index must be a nonnegative integer, and if it is 0, the value returned by the expression is the termination value. If it is nonzero, the function is applied to two arguments: the integer, and a copy of the original \(R\) expression with the integer decremented by 1. The result is the function composed with itself multiple times, with the index provided as an argument in each iteration. As an example, consider:

\[
\begin{align*}
R \text{ a t } f & \Rightarrow t 3 (R \text{ a t } f 2) \\
& \Rightarrow f 3 (f 2 (R \text{ a t } f 1)) \\
& \Rightarrow f 3 (f 2 (f 1 (R \text{ a t } f 0))) \\
& \Rightarrow f 3 (f 2 (f 1 a))
\end{align*}
\]
letrec \( f = (\lambda n (if \ (n = 0) 1 \ (\times n (f (- n 1)))) \)
\[ \rightarrow [n] (if \ (n = 0) 1 \ (\times n (f (- n 1)))) \]
\[ \rightarrow [S (S [n] (if \ (n = 0) [n] [n] \times n (f (- n 1)))) \]
\[ \rightarrow [S (S (K I) (S (S [n] [n]) (K 1) (S (S [n] [n]) [n] (f (- n 1))))) \]
\[ \rightarrow [S (S (K I) (S (S (K - 1) I) (K 1) (S (S (K x) I) (S ([n] [n]) (f (- n 1))))) \]
\[ \rightarrow [S (S (K I) (S (S (K - 1) I) (K 0)) (K 1)) (S (S (K x) I) (S (S (K [n]) [n])) (f (- n 1))))) \]
\[ \rightarrow [S (S (K I) (S (S (K - 1) I) (K 0)) (K 1)) (S (S (K x) I) (S (S (K [n]) [n])) (f (- n 1))))) \]
\[ \text{FIGURE 11.7} \]
Bracket abstraction on factorial.

<table>
<thead>
<tr>
<th>Combinator Name</th>
<th>Lambda Equivalent</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>((\lambda wxyzw(xyzw)))</td>
<td>S' M N O P → M (N P O P)</td>
</tr>
<tr>
<td>B - Composition Combinator</td>
<td>((\lambda x y z k y z))</td>
<td>B M N O → M (N O)</td>
</tr>
<tr>
<td>B'</td>
<td>((\lambda w x y z w x y z))</td>
<td>B' M N O P → M (N O P)</td>
</tr>
<tr>
<td>C - Reversal Combinator</td>
<td>((\lambda x y z k x y z))</td>
<td>C M N O → M N O</td>
</tr>
<tr>
<td>C'</td>
<td>((\lambda w x y z w x y z))</td>
<td>C' M N O P → M (N P O)</td>
</tr>
<tr>
<td>W - Repeat Combinator</td>
<td>((\lambda x y z k x y z))</td>
<td>W M N → M N N</td>
</tr>
<tr>
<td>Y - Fixed Point Combinator</td>
<td>((\lambda x y z k x y z))</td>
<td>Y M → M (Y M)</td>
</tr>
<tr>
<td>P</td>
<td>((\lambda x z k z))</td>
<td>P M N O → O M N</td>
</tr>
<tr>
<td>T</td>
<td>((\lambda x y z k x y z))</td>
<td>T M N → N M</td>
</tr>
<tr>
<td>I</td>
<td>((\lambda x y k x y k))</td>
<td>I M → I</td>
</tr>
<tr>
<td>Z - Zero or True</td>
<td>((\lambda x y k x y k))</td>
<td>Z M N → M Also Z = K I</td>
</tr>
</tbody>
</table>

1...Integers | \((\lambda n m n n (n x n) x n\) as above |

R - Primitive | \((\lambda x y z)\) | R M f Z → M |
Recursion | \((\lambda x y z)\) | R M f 1 → 1 (R M N (n - 1)) |

**FIGURE 11.8**
Some additional combinators.

If \( f(x, y) = x \times y \) and \( a \) is 1, the result of this is 3!. Thus, treating "\( \times \) as a prefix operator, we get that \( \text{factorial}(n) = R_1 \times n \), which gives an elegant caf form for \( \text{factorial} \) of simply \( R_1 \times n \).

**11.3.1 Some Interesting Combinator Expressions**
(Curry et al., 1972; Diller, 1988)

As an interesting insight into the power of these additional combinators, Figure 11-9 gives combinator-based expressions which function as we
Basic Function Equivalents:

\[ \text{cons} = B (C (C I) A_1 A_2) \]
\[ \text{car} = C I K \]
\[ \text{cdr} = = C I (K I) \]

Proof that car works: \(\text{car} (\text{cons}(A_1, A_2))\)

\[ = C I K (B (C (C I) A_1 A_2)) \]
\[ = I (B (C (C I) A_1 A_2) K) \]
\[ = B (C (I A_1) A_2) K \]
\[ = C (C (I A_1) A_2) K \]
\[ = C I A_1 K A_2 \]
\[ = K A_1 A_2 \]
\[ = A_1 \]

Proof for cdr: \(\text{cdr} (\text{cons}(A_1, A_2))\)

\[ = C I (K I) (B (C (C I) A_1 A_2) \]
\[ = I (B (C (C I) A_1 A_2) K) \]
\[ = B (C (I A_1) A_2) (K I) \]
\[ = C (C (I A_1) A_2) (K I) \]
\[ = C I A_1 (K I) A_2 \]
\[ = K I A_1 A_2 \]
\[ = I A_2 \]
\[ = A_2 \]

**FIGURE 11.9**
Combinator expressions for list functions.

would like the list functions \(\text{cons}, \text{car}, \) and \(\text{cdr}\) to function. What they actually do is immaterial to how they act in expressions based on them. For example, one could define the semantics of these three functions as a set of functions that obey the following properties:

- \(\text{cons}\) takes two arbitrary arguments.
- \(\text{car}\) and \(\text{cdr}\) take only one argument, and the result from that argument is undefined for anything other than an expression that is the cons of two things.
- For all expressions \(A_1\) and \(A_2\), \(\text{car}(\text{cons}(A_1, A_2)) = A_1\).
- For all expressions \(A_1\) and \(A_2\), \(\text{cdr}(\text{cons}(A_1, A_2)) = A_2\).

**FIGURE 11.9** proves that these combinator expressions have the desired properties.

As another example, the following combinator expression is the *successor function*, which returns one less than the single number presented as an argument, except for 0 which always returns 0:

\[
\text{successor} = C (C (C I (S (B (C (C I) A_1 A_2) (S B) (C (I (K I)))) (C I (K I)) (K I)))) K
\]

where \(\text{successor } n\) is-if \(n = 0\) then 0 else \(n - 1\). In this expression the

**FIGURE 11.4**
**OPTIMIZATIONS**

As can be seen from Figure 11-6, it is absolutely necessary to find ways to shrink a caf derived from an arbitrary lambda expression. That this might be possible should be obvious from the figure. There are multiple copies of identical subexpressions all over, such as \(S (K K (K S))\). Either we find shorter equivalents to such expressions, or we invent new combinators to take their place. While the latter is certainly feasible (cf. the supercombinators discussed later in this chapter), the former should be employed to the maximum extent first.

As an example of the power of such optimizations, consider an expression of the form \((S (K E_1) E_2)\), where \(E_1\) is an arbitrary caf. By simple reductions, we find that this expression is equivalent to the caf \(E_2\), alone (i.e., for any caf \(E_2\), \(S (K E_1) E_2 \Rightarrow (K E_1) E_2 \Rightarrow (I E_2) \Rightarrow E_2\)). Thus, whenever the compilation process finds that it has generated an expression of the form \((S (K E_1) E_2)\), it can substitute \(E_2\) in its place.

**FIGURE 11-10** gives a table of similar easily proven optimizations. If used extensively during the compilation of a lambda function into a caf, such optimizations can very dramatically reduce the size of the resulting caf. Figure 11-11 gives some examples based on Figure 11-4; Figure 11-12 gives a more powerful one—the translation of the successor function that blew up before. The result from this latter example is the simple caf \(S B\), something much more in tune with our notion of successor as a primitive function.

---

for any cafss \(x, y\)

1. \(S (K x) (K y) = K (x y)\)
2. \(S (K x) I = x\)
3. \(S (K x) y = B x y\)
4. \(S x (K y) = C x y\)
5. \(S (B K x) y = S' K x y\)
6. \(S x I y = W x y\)
7. \(S K x y = I x\)
8. \(B (K x) y = B' K x y\)
9. \(B x I y = x y\)
10. \(C (B K x) y = C' K x y\)
11. \(C K x y = I x\)

**FIGURE 11.10**
General combinator optimization rules.
As a proof that $S \ B$ does in fact implement the addition of 1 to an integer, consider that

$$Z_{i+1} a b \Rightarrow a(a(\ldots a(a b)\ldots)) \quad \text{with } i+1 \ a's$$

and that

$$(S \ B \ Z) a b \Rightarrow S \ B \ Z \ a b \Rightarrow B \ a \ (Z_a) \ b$$

$$\Rightarrow a(Z_a \ a \ b)$$

$$\Rightarrow a(a(\ldots a(a b)\ldots)) \quad \text{with } i+1 \ a's$$

Example: $(x + 1 \ x) = S(K + ) (K \ 1) \ 1$

$\Rightarrow S(K + ) (K \ 1) \ 1 \ \text{by Rule 1}$

$\Rightarrow + \ 1 \ \text{by Rule 2}$

Example: $(\lambda \ x \ y \ (x \ y)) = S(K \ K \ I)$

$\Rightarrow K \ \text{by Rule 2}$

Example: $(\lambda \ y \ x \ y \ (x \ y)) = (S \ (S \ (K + ) \ (S \ (K \ x) \ 1 \ 1)) \ 1)$

$\Rightarrow (S \ (S \ (K + ) \ (S \ (K \ x) \ 1 \ 1)) \ 1) \ \text{by Rule 2}$

$\Rightarrow (S \ (S \ (K + ) \ (W \ x) \ 1) \ 1) \ \text{by Rule 6}$

$\Rightarrow (S \ (B \ (W \ x) \ 1) \ 1) \ \text{by Rule 3}$

$\Rightarrow (W \ (B \ (W \ x) \ 1)) \ \text{by Rule 6}$

Check: $(W \ (B \ (W \ x) \ 1)) \ 3$

$\Rightarrow (B \ (W \ x) \ 3 \ 3)$

$\Rightarrow + \ (W \ x) \ 3 \ 3$

$\Rightarrow (x \ 3 \ 3) \ 27$

Note: Rule numbers refer to Figure 11-10. Examples should be compared with Figure 11-4.

**FIGURE 11-11**

Optimization of simple compiling.

$$(\lambda \ x \ y \ (x \ y)) = "\text{Successor Function}"$$

$\Rightarrow (x \ y) (S (K \ y) \ (S (K x) (y x)))$

$\Rightarrow (x \ y) (S (K \ y) \ (S (K x) (y x))) \ \text{by Rule 1}$

$\Rightarrow (x \ y) (S (K \ y) \ (y (x y))) \ \text{by Rule 2}$

$\Rightarrow (x \ y) (S (K \ y) \ (y (x y))) \ \text{by Rule 3}$

$\Rightarrow (x \ y) (S (K \ y) \ (x y) (x y)))$

$\Rightarrow (x \ y) (S (K \ y) \ (x y) (x y))) \ \text{by Rule 4}$

$\Rightarrow (x \ y) (S (K \ y) \ (x y) (x y))) \ \text{by Rule 5}$

$\Rightarrow (x \ y) (S (K \ y) \ (x y) (x y))) \ \text{by Rule 6}$

Thus: $(\lambda \ x \ y \ (x \ y)) = (S \ B)$ (compare with Figure 11-6)

**FIGURE 11-12**

A more complex example.

### 11.4.1 Recursion

Our first introduction to a lambda calculus implementation of recursion involved the $Y$ combinator. To make a function $(\lambda \ n \ E)$ recursive, we re-wrote it as

$$Y \ (\lambda \ f \ ((\lambda \ n \ E))$$

where inside $E$ we use $f$ for recursive calls.

Expressing a recursive function in combinators follows directly: we convert the body of the function to combinator form, then abstract out the function name and precede the whole expression by $Y$, namely:

$$Y \ [[f]([n]E)]$$

Figure 11-7 diagrammed the conversion of the body of factorial to combinator form. Obviously, one could extract $f$ from this as required above. Needless to say, this is painful. To again demonstrate the use of optimizations and shrink the expression, Figure 11-13 optimizes just Figure 11-7 using the $B$ and $C$ combinators. This figure also shows by example that applying the resulting expression to 3 delivers the expected results.

The next step is to abstract $f$ from this expression. Figure 11-14 diagrams the major steps of the process (we used some optimizations to be
proven as problems at the end of the chapter). The resulting combinator expression is about as long as the original lambda description.

11.5 GRAPH REDUCTION

A combinator expression consists of expressions of constants (combinators, built-in functions, numbers, ...). Reducing it involves finding something in the function position of an application and applying it. For functions that are combinators of arity n, this involves simply taking the next \( n \) subexpressions and rearranging them. When written as a linear string of characters (as has been done in all examples to this point), each reduction simply shuffles the character string form. For that reason it is often called string reduction.

Consider what might happen if we arrange such expressions as graphs where each subexpression is the root of a subgraph, with the leftmost arc of the subgraph pointing to the function subexpression, and the arcs to the right representing the argument expressions, in order. Then the leaves of the graph are all the constants in the expression (see Figure 11-15).

A graph reduction of such a graph involves finding any subgraph where the leftmost subexpression (the function) is a constant, and the right branches going back up the subgraph are its arguments, and replacing the entire subgraph by the result of the application. For function leaves that are combinators, this involves rewriting the subgraph with a new one where the arguments have been rearranged.

The following subsections discuss graph reduction in more detail.

11.5.1 Curried Graphs

Although the concepts behind Figure 11-15 seem simple enough, trying to develop a general algorithm that performs this automatically has a basic problem. All the arguments for any particular combinator may not be in the subgraph containing the combinator as its function. For example, Figure 11-15 was written so that all the arguments for the uppermost S (the ones labeled "a", "b", and "c") are at the same level of the same subgraph.

In most circumstances, however, it is possible for two, or even all three, arguments of S to be in subgraphs other than the one containing S. To see this, consider Figure 11-15(c). Here only two of the three arguments that C needs are in the same subgraph. The third, 3, is in the next subgraph up.

Needless to say, to devise an algorithm for automating such graph reductions would be somewhat messy. We would have to know the arity of the function and count the number of arguments in the same subgraph.

One way to avoid this complexity is to standardize our representation of applications back to that for the original lambda calculus, namely, every application has exactly one function expression and one argument expression. If the function can accept \( n \) arguments, the result of the first application is a curried function of \( n-1 \) arguments. This function expression can in turn accept exactly one more argument and produce a curried function of \( n-2 \) arguments. The process repeats until all the arguments are satisfied.

In terms of a graph, the original function (the leftmost constant) is
paired with the first argument in a binary graph. Depending on the complexity of this argument, the rightmost branch of this graph can point to either a constant itself or to a subgraph representing an expression.

If there is a second argument, it forms the right subgraph of a new binary graph, where the left subgraph is that constructed from the graph representing the curried function. The process repeats for each argument. Thus, for a function of n arguments, the corresponding graph is a binary graph of n levels, where each sublevel representing the function part is spawned from the left branch of its parent. Each right branch represents an argument.

Of course, if an argument of a subgraph is itself an expression, the equivalent right branch in the graph equivalent is a binary subgraph constructed according to identical rules.

Figure 11-16 diagrams the general case for an expression with n arguments. Figure 11-17 handles the specific case of the expression from Figure 11-15. Figure 11-18 diagrams yet another case, this one for a lazy evaluation.

To repeat, the key characteristic of graphs constructed from calfs in this fashion is that if one travels down the left branches of a subgraph until a constant is reached, then that constant represents a function. If the arity of that function is n, then the rightmost branch is its first argument, and the rightmost branches of the n – 1 parent nodes represent the other n – 1 arguments.

### 11.5.2 Subgraph Rewrite Rules

Any constant found as a left branch leaf can be treated as a function in some subexpression and as such is a candidate for reduction. We can actually perform it if there are enough arguments in the graph above it arranged as pictured in Figure 11-16. Reducing the resulting subgraph expression involves replacing the entire subgraph with a new subgraph representing the results.

\[ (f \text{ A1 A2 A3} ... \text{ An}) \]

\[ f = \text{function expression} \]

\[ A_i = i^{th} \text{ argument expression for } f \]

**FIGURE 11-16**

General n-argument curried graph.

For built-in functions this replacement is the value of the application. For combinators the replacement is often a new graph constructed from a reordering of the original argument subgraphs. Figure 11-19 diagrams some samples of these.

After either of these replacements we are free to look at the graph again for another function leaf to apply. This again involves diving leftmost until a constant is found.

### 11.5.3 Graphical Forms for Recursion

Recursion can be handled by the Y combinator as pictured in Figure 11-19(f). This is clean and straightforward, but suffers from the problem that one must abstract out of the body of the recursive function the identifier being used for the function's name. Again, this is not difficult, but it often leads to increased combinator code to reshuffle expressions and get the identifier out and its replacement actual argument back in.

An alternative form fits very naturally into our graphical representation (Figure 11-20). Instead of abstracting out the function name, we leave it in, but handle it differently in the graph. Basically, we connect
11.5.4 Evaluation Orders and a Reduction Algorithm

As with pure lambda calculus, there are a variety of ways to decide which reduction to do next in a binary combinator graph. In **normal-order reduction** we start at the root and go left at each internal node until we arrive at a leaf node. The constant at the leaf is the function to be applied. Picking off the right branches of the n parent nodes of this leaf provides the n arguments needed for its complete evaluation. After replacing this entire n − 1 level subgraph, the process is repeated.

In **applicative-order reduction** any leaf on the left of some internal node may be used for a function in an application. In particular, we would like to take those spinning off of the right branches for the arguments of a function chosen as above. Reduction is similar; the argument subgraph is replaced by the result of the evaluation.

The problem with the latter technique is that very often not all the arguments to a function are available yet, and if we were to replace that subgraph with its reduction, that replacement would have to be a curried application, which in combinator theory is the original expression itself! For example, the second C from the left of Figure 11-17 is in a left branch position of a subgraph, but if we go up the graph we find only two arguments identified ("=" and "0") before the subgraph dies. Replacing this subgraph is possible, but without expanding C into its lambda equivalent, doing what partial substitutions are possible, and returning a specialized lambda function [(x2 = z1) in this case], there is nothing to do other than return the original subgraph.

Besides these, the standard concerns about infinite loops and duplicate evaluations also hold.

Based on these comments, the standard algorithm for graph reduction is normal-order evaluation using a *left ancestors stack* (cf. Figure 11-21). We start at the root and go left. If the node there is not a leaf, we push onto the top of a stack a reference to that node, and go down and left one more level. The process is repeated until a constant leaf is reached. For reasons to be described later, what gets pushed on the stack is the node just investigated, not the argument found in its cdr.

At the point where the leftmost node is a constant, the top n references on this stack are to graph nodes whose right-hand branches refer-
ence the n arguments needed by the function. Further, the (n+1)-st entry on the stack is to the node whose right branch contains the first argument not needed by this application.

Figure 11-22 gives a set of abstract programs to perform such reductions on graphs built as s-expressions. The main function is \texttt{look-left}, which accepts two arguments, the current subgraph left to be traversed, and a list of parents of this subgraph. As long as the node is not a function constant, then the current node is pushed to the list, and \texttt{look-left} is called on the node’s left subgraph. When a constant is encountered, the function \texttt{apply} gathers up the proper arguments from the stack, reorders them appropriately into a new subgraph, and recalls \texttt{look-left} on the new subgraph.

Two auxiliary functions used to access the argument stack are \texttt{get-arg}, which gets the n-th argument from the stack, and \texttt{drop-arg}, which drops n arguments from the stack. The \texttt{get-arg} function includes a \texttt{cdar} of the stack to remove an argument. This reflects what was consed onto the graph. It corresponds to normal evaluation.

The stack generated inside of apply is built from the original stack of arguments with the n arguments consumed by the selected function deleted. In most cases, this new stack is given to another call to \texttt{look-left} for processing, along with the subgraph resulting from the application. \texttt{look-left} will look down through this subgraph until a new constant left leaf is found. During this pass, nodes holding arguments are pushed onto the stack as before. Thus, when it comes time to process the new function, there is no distinction as to where the arguments came from; they are all simply in order on the stack.

Notice also that for combinators no evaluation of arguments is performed; argument subgraphs are simply shuffled around as needed. This is not true for many built-ins, such as the \texttt{"+"} shown here. Both arguments must be evaluated using the same \texttt{graph-reduce} function before the replacement value can be computed.

\begin{verbatim}
graph-reduce(graph) = look-left(graph, nil)
look-left(root, stack) = "cons nodes to stack until leftmost
  an atom, and return resulting stack" =
    if atom(root)
      then if is-a-function-constant(root)
        then apply(root, stack)
        else root
      else look-left(left-branch(root), cons(root, stack))
apply(function, stack) =
  if function = \texttt{I}
    then look-left(get-arg(1), stack), drop-args(1)
  elsif function = \texttt{K}
    then look-left(get-arg(1), stack), drop-args(2)
  elsif function = \texttt{S}
    then look-left(cons(cons(get-arg(1), get-arg(3)),
                   cons(get-arg(2), get-arg(3))),
                   drop-args(3))
  else ... "handle other combinators and builtins"
get-arg(n, stack) = "get n^th argument from top of stack"
  if n = 1
    then right-branch(car(stack))
  else get-arg(n-1, cdr(stack))
drop-args(n, stack) = "drop n arguments from stack"
  if n = 0
    then stack
  else drop-args(n-1, cdr(stack))
\end{verbatim}

**FIGURE 11-22**

A left ancestor reduction algorithm.

### 11.5.5 Avoiding Argument Reevaluation

An interesting observation about the algorithm of Figure 11-22 is that it does not modify the actual data structures, whatever they are, that represent the graph. (Of course, as implied by the construction of the programs, an s-expression format where the left-branch is the car of a cons cell and the right-branch is the cdr is a natural fit.) The variable stack that is passed around is bound to a list of subgraph nodes representing intermediate points, but no attempt is made to go back and actually modify the branch fields of parent nodes when a subgraph is modified. In fact,
even though new subgraphs are created every time a function is performed, no prior graph is ever actually modified.

This approach exhibits all the advantages and disadvantages of normal-order evaluation. No argument is ever evaluated before it is needed, and the algorithm itself is easily verified as giving correct operation.

However, it is possible to evaluate the same subgraph more than once. This is because when we copy arguments, we are copying pointers to graphs that never change after they are built. Then, each time the value of an argument is needed, the corresponding graph is reevaluated afresh. This is almost equivalent to simply copying the argument graph as required. As an approach to sharing objects it is often called structure copying.

As an example, consider Figure 11-23. After the S reduction there are essentially two copies of the subexpression (+ 6 3). As reduction goes forward, each copy is evaluated separately.

Now consider what happens if, when subgraphs are first evaluated, we destructively modify them so that all future references to the same subgraph see the same evaluated result. This is exactly what we earlier termed lazy evaluation.

To implement this we must keep exactly one copy of each subgraph that will be modified as evaluation goes on and “share” this copy among all references to it. This is termed structure sharing.

Figure 11-24 diagrams as an example a potential representation of the expression (S E1 E2 E3). The cell marked with a “*” is the uppermost cell associated with the expression. After reduction, the new subgraph should be rooted in this cell’s parent cell field.

There are two ways of attacking the problem of substituting a new subgraph in there. First is to allocate a new cell and modify the pointer in the parent of the “*”-ed cell to point to the new graph. This does not work unless we also modify all other pointers to the old data structure. The second, and proper, choice is to leave the topmost cell of the subgraph in place, and modify its contents appropriately. All current references to the subgraph will thus in the future see the new reduced version. For the S combinator this involves allocating two new cells to hold pointers to the appropriate subexpressions (E1 E3) and (E2 E3), and re-writing the car and cdr of the “*”-ed cell to point to them.

Now, looking back at Figure 11-22, we see why the parent of an argument node, rather than the argument, is stacked. This allows access to node cells whose car should be modified.

For built-ins like “+”, a similar approach works. The equivalent of the “*”-ed cell in Figure 11-24 is modified to contain the final value of the operation. All future references will see it without any need for flags, recipes, etc.
There is one problem, however, with combinators like $I$ and $K$, which simply return one of their arguments. On the surface this sounds like "$+"," but the problem arises when the argument being returned is a pointer to another subgraph. A single pointer does not fit directly into a cell that was designed for a pair.

There are two solutions. First is to return $(I\ E)$, where $E$ is the argument being returned. This works, but it requires all sorts of special tests to handle the $I$ whenever it appears.

The alternative is to change the "$+"-ed cell to have a tag of invisible pointer with a pointer to $E$ as its value. This tag is essentially the same as mentioned earlier for Baker's garbage collection algorithm, and was used in several of the LISP machines we discussed. Although it uses up some extra storage, it is a much cleaner solution.

### 11.6 SUPERCOMBINATORS

(Hughes, 1982; Peyton-Jones, 1987; Diller, 1988)

The combinators discussed so far are relatively primitive, low-level functions that require a compiler to laboriously build up some combination that matches a particular expression. In contrast, a supercombining is a single combinator that is specially built by a compiler to match optimally the outer level of some function being compiled. When executed, the result is a greatly reduced number of data shuffles. Arguments are moved only when necessary, and then exactly into the positions desired.

Such a function represents a potentially huge increase in performance. Of course it does complicate the basic reduction algorithm. Code (sometimes microcode) must be built dynamically to handle each new supercombinator, and then loaded and invoked as needed.

The following subsections address a more formal definition for supercombinators, a description of the key objects looked for by the compiler—namely, maximally free expressions—optimal orderings of arguments for these supercombinators, and some other optimizations.

#### 11.6.1 Defining and Finding a Supercombinator

Formally, the expression $(\lambda x_1 \ldots x_n(M))$ is a supercombinator of arity $n$ if:

- $M$ is not a lambda function;
- And any subexpression of $M$ which is a lambda function is itself expressible as a supercombinator.

This means that $M$ is a caf involving only built-ins, combinators, other supercombinators, and the identifiers $x_1 \ldots x_n$. Applying the function $(\lambda x_1 \ldots x_n(M))$ to a series of $n$ argument expressions will then result in a shuffling of those expressions, with additional constants thrown in. This is again a caf.

There are some important differences between a supercombinator and a regular combinator. First, these supercombinators are built from user-provided functions, not defined beforehand. Next, unlike the prior combinators, the result of applying a supercombinator can explicitly insert new constants into the expression (in fact, most real supercombinators introduce several). Finally, optimized supercombinators order the arguments they process to delay as long as possible their evaluation and maximize their potential reuse. This will permit a very useful form of lazy evaluation.

A very simple example of a supercombinator is:

$$\alpha = (\lambda z_1 z_2) \overset{=} {=} (z_1 z_2)$$

This caf takes four arguments, compares the first two, and then selects one of the remaining for continued evaluation. Its rewrite rule is:

$$\alpha \ M_1 \ M_2 \ \Rightarrow \ \{= M_1 \ M_3\}$$

Its typical use in a caf for some specific function might look like:

$$(\alpha \ 0 \ n \ (f \ (- \ n \ 1)))$$

As another example, we might build a supercombinator of three arguments with the following rewrite rule:

$$\alpha \ M_1 \ M_2 \ M_3 \ \Rightarrow \ \langle M_1 \ (\text{car} \ M_2) \ (\text{cdr} \ M_3) \rangle$$

The basic algorithm for finding such supercombinators starts with the function being compiled, such as

$$(\lambda x_1 \ldots x_n(M))$$

and works inside out, starting with $x_n$ and working backward to $x_1$. At each step, the innermost expression $(\lambda x_1(M_k))$ has a new supercombinator built for it, and the result is used to construct an $M_{k-1}$ caf for the next iteration. For each such supercombinator, the equivalent rewrite rule must be constructed and saved for use when the function is evaluated.

The actual steps in converting $(\lambda x_1(M_k))$ to a caf called $M_{k-1}$ involving a new supercombinator $\alpha_{k-1}$ are as follows:

1. Find all the subexpressions of $M_k$ that are independent of $x_k$. These are the ones that have no free occurrences of $x_k$ in them. They are called free expressions.
2. Pick the most encompassing of these expressions (those not part of other free expressions) as the maximal free expressions. Assume that they are $N_1, \ldots, N_m$.
3. Invent $m$ new identifiers $z_1, \ldots, z_m$ not in use anywhere.
4. Define supercombinator \( \alpha_k = (\lambda z_1 \ldots z_m \cdot x_k) [z_1/N_1, \ldots, z_m/N_m]M_k \).

5. Until \( x_i \) has been handled, repeat the process with \( (\lambda x_{k-1} \cdot M_{k-1}) \) where \( M_{k-1} = \alpha_k N_1 \ldots N_m \).

Figure 11-25 diagrams the standard example of this process used by the chief inventor of supercombinators, John Hughes.

### 11.6.2 Free Expressions

The key step in the above algorithm is the location of the **free expressions** in \( M_k \). These expressions are independent of \( x_k \), and thus can be “factored out” of the expression that will be converted into a supercombinator \( \alpha_k \) and made into an argument for it.

Formally, \( N_i \) is a free expression of \( M_k \) if both

- \( N_i \) is a subexpression of \( M_k \).
- The value of \( N_i \) is guaranteed to be independent of \( x_m \), that is, a caf involving only constants, built-ins, of the identifiers \( x_1 \) through \( x_{k-1} \).

For example, the free expressions in

\[
(\lambda s (\text{if} (= n 1) \text{(car} s \text{)} (\text{f} (- n 1) \text{(cdr} s \text{)}))
\]

are (in rough order of size)

- if, -, n, 1, car, f, -, n, cdr
- (= n), (- n)
- (= n 1), (- n 1)
- (if (= n 1)), (f (- n 1))

Note that \( (\text{f} (- n 1) \text{(cdr} s \text{)}) \) is *not* a free expression because it is not independent of \( s \). Also, the last of these free expressions are not subexpressions of any bigger free expressions, and are thus **maximally free expressions** (we do not count simple constants as maximally free).

### 11.6.3 Identifying Maximally Free Expressions

There are several algorithms for identifying maximally free subexpressions from an arbitrary expression. One of them in particular is both simple and will help in some later optimizations. In this algorithm we assume that the identifiers in the overall function being compiled are read from left to right, and are each given a number (called its **label**) to be used in further processing. Thus, in \((\lambda x_1 \ldots x_n \cdot M)\), \( x_i \) is assigned a label of \( i \).

Basically, the algorithm uses these labels to assign a number (called its **level**) to each subexpression \( N \) of \( M \) as follows:

- If \( N \) is a constant or built-in, assign it a level of 0.
- If \( N \) is an identifier \( x_i \) with label \( i \), assign it a level of \( i \).
- If \( N \) is an application, assign it a level equaling the greater of the levels of the function and argument subexpressions.

Figure 11-26 diagrams a sample assignment. In general, any subexpression with number less than \( k \) is independent of the identifiers labeled \( k \) or higher. Thus, when processing \((\lambda x_k \cdot M_k)\), any subexpression whose level is less than \( k \) and whose next bigger expression has level \( k \) is a maximally free expression.

---

**Figure 11-25**
Hughes' original supercombinator example.

**Figure 11-26**
Sample numbering.
11.6.4 Ordering the Maximum Free Expressions

In Figure 11.25 there were two subexpressions factored out and made arguments of the supercombinator \( \alpha_3 \). Notice that to this point the order of these arguments has not been governed by any particular functional concerns. We could have derived \( \alpha_4 \) from

\[
(\lambda z_1 z_2 z_3 f \ (z_2 \ (\text{car} \ s) \ (z_3 \ (\text{cdr} \ s)))) \ (f \ (- \ n) \ 1) \ ((f \ (= \ n) 1))
\]

What difference is there in these two orderings? In the first one both arguments are functions of \( n \), while only the first argument is still dependent on \( f \). This is reversed in the second formulation.

In general, assume that we have

\[
(\lambda x_1 \ldots x_{k-1} \ \alpha_k N_1 \ldots N_i \ N_{i+1} \ldots N_m)
\]

For any \( j \) we want \( N_i \ldots N_j \) to be all, and only, maximally free expressions for \( x_j \) in particular for \( k=2 \). This will lump all, and only, those expressions that are dependent on \( x_{k-1} \) to the far right, where they can be most easily dealt with. More precisely, we want to order the \( N_j \)'s so that for each \( x_j \) all \( N_j \)'s that depend on it occur before any that depend on any \( x_{j+1} \) through \( x_{k-1} \).

If the labeling algorithm described above for maximally free variables has been used, we can use these same numbers to order them. Basically, we sort the maximally free subexpressions and arrange them as arguments to the eventual supercombinator by level, with the lowest to the left. Once this ordering has been imposed, the use of the \( z_j \)'s inside the body of the supercombinator is specified.

One further optimization will occur if, within this set of maximally free subexpressions, there is exactly one at some level, and that one is a simple identifier. This will simplify some combinator later on by the application of a gamma optimization, namely,

\[
((\lambda x_1 \ldots x_i \ \alpha_k N_1 \ldots N_{m-1} \ x_i)
\]

can be replaced by

\[
((\lambda x_1 \ldots x_{i-1} \ \alpha_k N_1 \ldots N_{m-1})
\]

11.6.5 Completing the Example

Figure 11.27 summarizes all these steps as applied to Hughes' demonstration.

11.7 COMBINATOR MACHINES
(Clarke et al., 1980; Trelleven et al., 1982)

Combinators are for the most part simple enough to correspond to basic instructions in a more or less conventional computer. This section describes several attempts to implement such machines. Although their impact has been primarily on the advanced research community, there is significant promise, particularly for the G-Machine, and it would not be surprising to see offshoots show up in future high-performance machines.

11.7.1 The SKIM Machines
(Turner, 1979b; Stoye et al., 1984)

The first attempt to use combinators as the basic instruction set architecture of real hardware began in the late 1970s with the remodeling of a conventional computer to perform graph reduction using the left ancestor stack reduction algorithm. The basic instructions were \( S, K, \) and \( I \), plus the other major combinators of Figure 11-8. This implementation was followed by another that used list pointer reversal instead of a separate stack of pointers for the arguments.

For obvious reasons these machines were called the SKIM machines.

SKIM I The original SKIM machine was an existing conventional computer that had writable microcode control store. A left ancestor stack graph reduction interpreter was written in this microcode, along with direct microcode support for the basic combinators \( S, K, I, Y, C, B, U \), and the standard integer functions for arithmetic, comparison, etc. A compiler translated programs written in SASL (a functional language similar to our abstract language) into optimized combinator code suitable for loading into this machine.

The memory for the machine looked much like our standard list-oriented memory. Memory was a pool of cells that could be either a single integer or a pair of pointers. A program consisted of a graph built from such cons cells, where in each cell the car points to the function...
subexpression and the cdr points to its first argument subexpression. A function corresponding to a combinator or built-in was represented by an integer which specified an appropriate microcode routine.

Execution involved dynamically modifying the memory representing the program structure. The basic microcode kept a stack (in memory) of pointers back into the current program. When a combinator was found as the leftmost object, the matching microcode was executed, leaving the program graph modified exactly as described earlier.

Copies of arguments were made by structure sharing, i.e., copying just a pointer to the original subgraph for an argument. This permitted an easy lazy evaluation, since once an argument was evaluated, its subtree was replaced by the reduced value. All other pointers to that subgraph would then see the reduced value without further work. As discussed previously, some of the combinators, such as I or K, required special attention to permit proper operation. This attention was by replacing roots of affected subgraphs by either the (I."value") cell or by an invisible pointer.

Performance of this system was analyzed using several benchmarks (see Figure 11-28). Comparisons were made to a conventional SECD Machine implementation. If we assume that a basic SKIM instruction is approximately as time-consuming as a basic SECD instruction (a rational first approximation), then, as shown, the SKIM machine seemed to take about twice as many instructions as the SECD implementation. If, however, we also look at the number of memory cells allocated dynamically by the programs during execution, a different result emerges. The SKIM machine often needs only half the cells needed by the normal SECD Machine, and up to one-tenth the number of cells needed by a fully lazy version. These latter numbers are of particular relevance because the SKIM implementation is inherently fully lazy. It simply does a better job of managing storage by dynamically modifying and reusing storage during execution. This translates into fewer garbage collections per program execution, and thus a significant performance advantage that is not obvious from the raw instruction counts.

SKIM II The success of the original SKIM experiment led to a second version that was also microcoded, but which was faster, with a bigger address space, more basic data types, etc. The major conceptual difference was in how the arguments were tracked. Instead of a stack of pointers to parent subgraphs, this machine simply reversed the car pointers in the graph as it went left and down. To handle this it kept two key pointers in machine registers. The forw register points to the cell containing the current function expression. The back register points to the cell whose car initially pointed to the forw cell (the parent of the forw cell).

When a function is found which is a subexpression (i.e., forw points to a cons cell), the following sequence of register transfers takes place:

1. temp ← car(forw)
2. back ← replace(forw, back)
3. forw ← temp

This process reverses the pointer in the car of the function expression, making that cell the car of the argument stack. forw receives the original contents of this cell so that a continued left and down scan can be performed. The cells joined backward from the back pointer match exactly the cells on the standard left ancestor stack.

When a combinator or built-in is reached, the arguments for it can be found by going backward through the back register, much as with the basic stack algorithm, but without the extra memory overhead of the stack cells, and without the extra indirect reference through the stack cells to get to the actual program cells. Figure 11-29 diagrams a sample of this for the S combinator.

Another architectural innovation in SKIM II was the addition of a 1-bit reference count field to each cell. This count indicates whether or not the pointer stored in a field is the only pointer to the designated cell. It is set to 0 when the cell is initially built, left alone when a combinator simply rearranges its arguments with one copy of an argument, and set to 1 when a combinator copies an argument into more than one location.

The advantage of this is that the microcode for each combinator can quickly determine if a cell becomes free during a rewrite, and thus can be immediately reused without having to go through a storage reclaiming process. In actual programs this technique was highly successful, with 70 percent of the memory cells reclaimed directly by the combinator code without recourse to a garbage collector.

In overall performance these improvements gave something like a 4-to-1 increase over SKIM I, putting it in a very competitive position with conventional computers.

11.7.2 The CURRY Chip
(Ransdell, 1986)

An example of the real simplicity that is possible when a machine is designed from the ground up to support combinators is the CURRY chip. This chip was part of a three-chip set (see Figure 11-30) built out of what
is today very-low-density VLSI. In approximately 9000 transistors, a complete CPU chip was built which performed as its instruction set basically the combinatorics accepted by the SKIM machines, but without microcode. A separate chip handled memory management and garbage collection in a fashion very similar to that for the SCHEME-79 chip described in Chapter 10. A third simple chip handled overall timing and interface with a test and control console.

At the slow (by current standards) clock rate of 0.4 MHz, the chip could perform about 18,000 reductions per second. This translates to about 13 machine cycles per instruction, which again is not bad at all for 9000 transistors.

The CURRY chip contained two registers, fun and stack, which correspond directly to the next and back registers of SKIM II. Keeping track of arguments is done via pointer reversal in the program graph.

FIGURE 11-29
Graph reduction with pointer reversal.

* These cells are modified.
+ These cells are new.
All other cells are unchanged.

(1) At start.
(2) When S found.
(3) After reducing S.

All functions (programs) for the CURRY chip have to obey the convention that they accept an input stream as their single arguments and deliver a single output stream as a result. Both of these streams are 1 bit wide, with 1-bit I/O paths connecting the CURRY CPU to the control chip, from which I/O to the outside world is actually performed.

The R combinator performs the basic read of 1 bit. When expressed as a rewrite rule, this combinator would be used as follows (where E is the code for the function to receive the input):

\[ R \ E \Rightarrow \]

\[ P \ K \ (R \ I) \ E \Rightarrow E \ K \ (R \ I) \] if input is a 0

\[ P \ J \ (R \ I) \ E \Rightarrow E \ J \ (R \ I) \] if input is a 1

The code in E would accept as its first argument either K or J which would then internally select one of two expressions for evaluation. These expressions would match the case for an input of 0 or 1, respectively. The other argument to E, (RI), basically provides a stream function for the next bit.

Output is similar. The expression (P KE) outputs a 0 when E is a stream function, while (P JE) outputs a 1.

Finally, code for the CURRY chip was also produced with a real dash of simplicity (see Figure 11-31). Programs written in the CHURCH language were translated by a C program into something very akin to a pure lambda calculus form. This was then processed by a 750-line CHURCH program into a CURRY program, using the standard bracket abstraction algorithm plus a handful of optimizations. This program was
11.8.1 Major Data Structures

There are several major data structures for the G-machine:

1. A value stack contains the results of fully evaluated expressions that are to be processed by built-in functions. The value stack is a simple stack formed from consecutive locations, with no tags on cells.
2. A pointer stack (similar to the left ancestor stack) points backward into the graph at the arguments. This data structure is again a simple stack.
3. A heap holds the often dynamically built expression subgraphs, referenced by the pointer stack.
4. A dump holds return information for recursive calls to eval, namely, pairs of pointer stack and code addresses.
5. The program code area consists of a sequentially organized set of G-machine instructions.

Some implementations also define an environment to store operands, although for the most part the pointer stack takes its place.

The key data structure is the pointer stack. When a subgraph is being evaluated, its list of arguments is pushed on the stack as with graph reduction. The elements of this stack are pointers to tagged cells in the heap. Instructions exist to modify (as with rplacd) entries in this stack, permitting lazy evaluation techniques to change a pointer once and have many references to an object see an evaluated equivalent.

The value stack is a very conventional stack used to perform basic arithmetic and the like in a more efficient manner than for the SECD Machine. Objects here are untagged and are found in consecutive locations. Instructions exist to move them to and from the pointer stack, adding and deleting tags as necessary.

The heap is organized similarly to the list memory used throughout our previous machines. Cells are allocated from it as needed, with pointers to them embedded in the other major memory areas. Each cell contains a tag field and either one or two value fields. Unlike our previous organizations, however, the tag field often is more than a few bits instead it is an address to some native machine code that performs some specific function, such as a built-in. Specific examples include tags of INT, BOOL, CONS, AP, FUN, and HOLE.

Cells of the first three types are like those in the SECD Machine. A cell with the fourth tag represents an Application, with its first value subfield pointing to a subgraph for the function and the second pointing to the argument subgraph. A cell with the tag FUN has one value field which points to the code for some FUNCTION (built-in or supercombinator). The final tag, HOLE, has a value field representing a holder to be filled in later. It is used to construct cyclic data structures as is done with the SECD DUM instruction.

The G-machine’s native instructions are stored in consecutive locations as with most conventional machines.
11.8.2 Typical Instructions

Figure 11-32 gives a brief description of some of the G-machine instructions that manipulate these data areas. The basic goal of these instructions is to manipulate a graph that is built on the heap. Instructions of the form MKx then dynamically build subgraphs consisting of cells with AP and FUN tags. These subgraphs are cons-like s-expressions whose car and cdr equivalents point to either other cells or compiled machine code for the body of a function.

The EVAL instruction is the most complex of the instructions, and does a leftmost dive through the topmost subgraph on the pointer stack, much as described earlier. Arguments are pushed onto the pointer stack until a basic function is found. The code for that function (as found in the cell with tag FUN) is then entered for execution. Return information is placed on the dump to handle nested calls and recursion.

Dynamically building code permits both curried functions and opens the door for lazy evaluation. Another instruction, UPDATE, completes this capability by replacing a prior argument subgraph with an evaluated result. All future references to this argument then see the computed value.

11.8.3 Example

As an example, Figure 11-33 lists the code for factorial. Initially we assume that the top item on the pointer stack (item 0) is to the n argument. This item is evaluated and compared to 0. If equal, that subgraph node is replaced by a "1." If not, a new subgraph is built, corresponding to $n \times \text{fact}(n-1)$. This is then evaluated and the result rewritten as before.

In real systems a compiler converts a function into G-machine

```
fact(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{else if } n = 0 \\
1 & \text{else if } n > 0 \\
\text{result} & \text{else}
\end{cases}
```

G-MACHINE CODE: At entry, element 0 on pointer stack is n.

```
fact: PUSH 0 ;Push 2nd copy of n to pointer stack EVAL ;Evaluate top copy PUSHINT 0 ;Push a 0 to value stack GET ;Transfer evaluated n to value stack EQ ;Compare top pair of value stack JFALSE L1 ;Jump if not equal
; Continue here if n = 0—pointer stack 0 = original n.
PUSHINT 1 ;Push result 1 to value stack L2: UPDATE 1 ;Do graph rewrite of original n RET 1 ;Return, popping top element (1)
; Come here if n > 0—pointer stack 0 = original n.
L1: PUSH 0 ;Push a copy of n back on pointer stack EVAL ;Evaluate it GET ;Transfer it to value stack PUSHINT 1 ;Push a 1 to value stack PUSH 1 ;Push a copy of the n argument PUSHINT sub ;Push a pointer to subtract MKAP ;Build subgraph code for (n - 1) MKAP ;Build subgraph code for ((n - 1) - 1) PUSHFUN fact ;Push a pointer to subtract MKAP ;Build subgraph code for (fact (n - 1)) EVAL ;Evaluate the subgraph GET ;Transfer result to value stack MUL ;Multiply n by fact(n - 1) MKINT ;Move result back to pointer stack JUMP L2 ;Again modify the appropriate argument.
```

**FIGURE 11-32**

Some typical G-Machine instructions.

**FIGURE 11-33**

Sample code for G-Machine.
code, often with optimizations similar to those for the SECD machine. From here, each of the G-machine instructions is expanded into an appropriate sequence of the native machine instructions for the target machine the code should run on.

11.8.4 Performance Studies

Some early performance data is encouraging. An early software-based system that generated fully lazy, VAX machine code (Johnsson, 1984) was tested over several benchmarks (Fibonacci, prime numbers, and insertion sort) and gave execution times within a factor of 2 (both slower and faster) of several other compiled versions, including a C-based one. Simulation of a specialized machine design which executes G-machine instructions directly (see Kieburtz, 1988) gave performance similar to that of a 3.6-mip computer with a relatively moderate machine cycle time and machine design.

Extensions of the G-machine to handle parallel graph reduction also have been demonstrated on an 18-way parallel processor (George, 1989). The major departures were in:

- The addition of a task pool to list runnable subgraphs
- No value stack—intermediate computations were kept on the equivalent of the pointer stack
- A flag used to determine if there are sufficient subgraphs to be worth spinning off parallel evaluation

On several benchmarks, performance was within a factor of 4 of that for a good compiler for a nonparallel, but largely equivalent, functional language running on a conventional sequential computer six times faster than the parallel one.

11.9 PROBLEMS

1. Apply the basic bracket abstraction process to the functions not, and, and or from Chapter 5. Assume that T and F are available as basic constants. Show by example that each translated sequence works as expected.

2. Compile \( \lambda \) into basic combinators, and check by applying to 3. Compare the code with Figure 11-4.

3. Repeat Problem 2 assuming that T and F are not available as built-ins.

4. Convert Figure 11-3 to a version that processes s-expressions of the form of Figure 6-5.

5. (Project) Upgrade the solution to Problem 4 to permit multiple arguments and assume built-in constants for numbers, booleans, standard arithmetic, and if.

6. Find pure lambda calculus equivalents to the combinator forms of car, cdr, and cons.

7. Show that the following are also valid definitions for car, cdr, and cons:

cons = (C(B C(B (C 1) K)))
car = (C 1(K 1))
cdr = (C 1(B W(B B)) (K 1))

8. Express the primitive recursion combinator R (see Figure 11-8) as an abstract program.

9. Translate the following into combinator, optimized combinator, and super-combinator form:

a. \( \lambda m \lambda n \lambda f f(n) \)
b. The lambda definition of addition

10. Show that the following equalities are true for any expressions X and F:

a. \( S F I = W F \)
b. \( S F (K X) = C F X \)

11. Using only the combinator R, the lambda definition of integers, and a "primitive" function \( \mathbb{1} = x \cdot 1 \), write a lambda or combinator expression for the boolean function \( \text{"true"} \).

12. Show all the steps involved in converting the first of the following definitions of factorial into the second. Then apply 3 to the second and derive the result.

Y \( \lambda S \lambda I \lambda K \lambda x (K x) \) \( I \)

Y \( \lambda S \lambda I \lambda K \lambda x S(K x) S(K(K x) S(K x)) \)

Y \( \lambda S \lambda I \lambda K \lambda x (K x) \) \( I \)

13. Convert the definition of member to combinator form. Assume that primitives such as eq, null, car, cdr, etc. are available and do not need to be abstracted further. Try out your function on arguments ‘a’ and ‘b < c’. List all steps.

14. Prove the following, assuming that E has no free occurrences of n in it, but that A might:

- \( A [E = (K E)] \)
- \( A [E = (S E A)] = (B (S E) [n]A) \)

15. Draw the binary graph form of Figure 11-12, and show the graph reductions until just before the start of the first recursion.

16. Compute an optimized combinator form for each of the following, and show the resulting binary graph:

a. \( \text{Fib}(n) \) where \( \text{Fib}(0) = 0, \text{Fib}(1) = 1, \text{Fib}(n + 1) = \text{Fib}(n) + \text{Fib}(n - 1) \)

b. \( \text{Ack}(3,3) \) where \( \text{Ack} \) is Ackerman's function, \( \text{Chapter 1} \)

17. Write a version of apply from Figure 11-22 that performs lazy evaluation by list modification as pictured in Figure 11-24. Make sure you point out how you handled combinators like K and I, and built-ins like ‘+’

18. Convert the supercombinator \( \alpha \) with the rewrite rule \( \alpha \text{A B C} \Rightarrow (A \text{car C}) (B \text{cdr C}) \) into a lambda function. Under what conditions might such a function be useful?

19. Convert the combinators \( \alpha_1 \) and \( \alpha_3 \) of Figure 11-25 into pure combinator form. Assume that the built-ins car, cdr, etc. are available.

20. Convert the following functions to supercombinator expressions:

a. \( \lambda x y z (x y z (x y z)) \)
b. \( \lambda x y z (x y z (x y z (x y z (x y z)))) \)

c. \( \lambda x y z (x y z (x y z (x y z (x y z)))) \)

21. How would you compare the complexity of the G-machine to that of the SECD machine? Include consideration of memory management.