S/K/ID: Combinators in Forth

Johan G. F. Belinfante

School of Mathematics
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

This paper describes using Forth as a metalanguage to construct a tiny, but extensible, functional programming language S/K/ID. In this type-free language both programs and data are combinators. A defining word `DEF:` is introduced to implement combinators as words in Forth. The core of S/K/ID consists of primitive combinators `S`, `K` and `ID` defined directly in Forth. The rest is built up in bootstrap fashion by a sequence of definitions. The construction of S/K/ID requires only a primitive Forth system. No control words are used. The execution of combinator code causes silent words to be written to the dictionary. To avoid using the `STATE` variable, these silent words are compiled using `LIT` and `,` . The code presented has been tested on a Commodore 64 computer using 64FORTH by Human Engineered Software, Inc.

Introduction

Much of the power of Forth derives from its extensibility. Starting from a small core vocabulary, the rest of the language is built up by a sequence of definitions. Just how small can the core vocabulary be? Perhaps the ultimate answer to this question is given by the theory of combinators. The core vocabulary may be reduced to two primitive combinators plus a few extra words, including an executor, a defining word and a garbage collector.

We develop such a combinator-based language using Forth as a metalanguage. Forth is used to define the core words and for all input/output and some test procedures. Some first steps in extending the core are taken to illustrate the general procedure.

Because the core is small, care must be taken to generate compact and efficient code. In several cases we have adopted unconventional definitions of standard combinators in an effort to produce simpler code. In this effort we were guided by a certain rough measure of complexity introduced in *Reducing Complexity*. The code presented here has been tested on a Commodore 64 computer using 64FORTH by Human Engineered Software.

Implementing combinators in Forth is fairly easy because Forth allows direct access to the compiler. The execution of combinator code causes many silent words to be automatically written to the dictionary. To avoid fussing with the `STATE` variable, these silent words are compiled using the words `LIT` and `,` rather than with `COMPILE` and `[COMPILE]`.

Are combinators actually useful? Combinators are currently finding some use in certain experimental fifth-generation programming languages such as Landin's SECD and the functional programming languages FP and HOPE ([LAN64]; [CLA80]; [BOE82]; [EIS85]). In addition some older programming languages, especially LISP, contain features which were inspired by the lambda calculus.

Combinators are an alternative to the lambda notation ([STE72]; [BAR84]).The relation of combinators to lambda calculus is similar to the relation between assembly language and higher level programming languages. An advantage of combinators over the lambda calculus is that no variables

are needed, resulting in economy in computer code. Combinators, however, are less familiar to most people and are perhaps less easy to use than the lambda calculus. In practice one may wish to write programs in the higher level lambda calculus using a compiler to produce combinator code [TUR79a, b]. A delightful account of combinators is presented in Smullyan's fascinating book, where all the combinators are given names of birds [SMU85].

Functional programming languages are not especially advantageous on today's computers with their von Neumann–style architecture because they often require more memory than conventional languages. The single-assignment feature of functional languages, however, may well make them more important in the future on machines with data flow architectures ([BAC78]; [JOU85]; [SHA85]). One may be able to trade spatial complexity for a gain in speed.

History of Combinators

Credit for inventing combinators goes to Moses Schoenfinkel who lectured about them before the Goettingen Mathematical Society on 7 December 1920. His aim was to reduce the number of primitive undefined notions needed for symbolic logic to as few as possible. Sheffer had already shown in 1913 that a single connective suffices for the propositional calculus. Schoenfinkel also showed that the predicate calculus could be simplified by eliminating all variables. The fundamental idea which makes this simplification possible is the admission of functions as arguments [SCH24].

Using variables to express logical properties lacks a certain economy of thought. Indeed, a statement in formal logic such as the tautology

A AND B = B AND A

is not really meant to be a statement about the variables A and B but is rather intended to be a statement about a symmetry property of the word AND. Eliminating variables from symbolic logic reduces the number of primitive notions needed for logic and thereby simplifies its foundations.

Schoenfinkel found that only the two combinators `K` and `S` are needed to eliminate variables from all logical statements. The combinator `K` is used to create constant functions. The combinator `S` is slightly more complicated. The name `S` stands for the German word *Verschmelzung*, which means "smelting." Curry suggested that `S` be called the distributive combinator. Other combinators can be defined in terms of these two. In particular, Schoenfinkel showed how to express the combinators `B` and `C` in terms of `S` and `K`.

Haskell B. Curry made the first serious attempt to base logic on combinators in his 1928 thesis. He also proved that `S` could be replaced by the conceptually simpler combinators `B`, `C` and `W`. The combinator `B` is used to form composites of functions. The combinator `C` is used to exchange the arguments of a function of two variables. The combinator `W` is used to equate the arguments of a function of two variables [CUR30].

Alonzo Church introduced the lambda calculus at about the same time, and it was shortly proved that the theory of combinators is equivalent to a version of the lambda calculus. (The original version of the lambda calculus omitted constant functions, but this is not essential.) Church showed how the arithmetic of natural numbers 0, 1, 2, ... can be developed directly in the lambda calculus [CHU33]. To do so we must think of numbers as functions. The number 2, for example, may be identified with the function whose value at *f* is the composite of two copies of *f*. These Church numbers, or iterators, as well as a successor combinator can be defined in terms of the primitive combinators.

Stephen Kleene completed this picture of arithmetic in 1932 by showing how to define the predecessor function within the lambda calculus [KLE80, p. 1]. The definition of the predecessor function in terms of combinators can be accomplished by using a pairing function and iteration. We shall do it a little differently to make the code more compact, but the basic idea remains the same. Going further, Kleene and Church were able to show that the class of lambda-definable functions of natural numbers coincides with the class of general recursive functions ([KLE36]; [PET81]; [KLE81]).

Despite these successes of the lambda calculus in providing a foundation for ordinary arithmetic, some problems remained. Kleene and Rosser discovered a paradox in the original version of the lambda calculus that for a time cast serious doubt on the entire enterprise [KLE35]. Cardinality objections also were raised, based on the idea that there are more functions than arguments for any set-theoretic universe. The cardinality objection can be avoided by Church's proposal that a combinator be interpreted as a function in the sense of a rule of correspondence rather than as a function in the sense of the collection of ordered pairs making up its graph [CHU41]. In modern times we may like to think of such a rule of correspondence as representing some computer code or its equivalent.

The consistency and nontriviality of the modern version of the lambda calculus and the theory of combinators has been established by the discovery by D. Scott and others of set-theoretic models of the lambda calculus ([SCO72]; [SCO77]; [STO77]; [SCO80]). These models have certainly led to an increased understanding of the meaning of the lambda calculus. This enterprise has recently been extended to the typed lambda calculus, influenced in many ways by ideas from category theory ([LAM80]; [LAM86]).

Basic Notation

The fundamental operation in the theory of combinators is that of applying a function to its argument. A careful analysis of the process of application is absolutely essential for understanding combinators.

The notation for functions in Forth differs from that used in mathematics. In mathematics the value of a function f for an argument x is denoted $f(x)$, but in Forth one usually writes `x f`, omitting the parentheses and using reverse Polish notation. A type distinction is implicit, as the stack pictures for `x` and `f` are not the same. Indeed, the word `x` puts something on the stack `( --- n1)`, whereas `f` transforms it: `(n1 --- n2)`. For example, consider the Forth expression `0 1+`. The word `0` puts zero on the stack, and `1+` takes zero from the stack and replaces it by one.

Ordinarily no problem arises from this distinction between arguments and functions, but it becomes awkward if we want a function to be an argument for another function. Combinators provide a functional programming style in which there is no type distinction between a program and its data. Combinators serve as both functions and arguments.

Another problem arising from the omission of parentheses in the Forth notation for application stems from the fact that the process of applying a function to its argument is not associative. An expression like `x g f` in Forth is used when we want to apply `g` to `x` and then apply `f` to the result. It would be awkward, if not impossible, to deal with the situation where `f` is to be applied to `g` and then the result applied to `x`. To prevent any misunderstanding on this point, we hasten to remark that the nonassociativity of application to which we refer here has nothing at all to do with the well-known associative law for composition. We shall have more to say about composition later.

In mathematics one usually resorts to the device of inserting parentheses to cope with the notational problems arising from nonassociativity. We would perhaps use parentheses to distinguish these two cases by writing $(xg)f$ and $x(gf)$. It is suggestive to imitate this in Forth by introducing the parentheses as new Forth words, but, in fact, we need only introduce the right parenthesis as a word to represent the process of application.

In S/K/ID we shall use the notation

```
x f )
```

for the result of applying `f` to `x`. The two different orders of application distinguished above can now be written as

```
x g ) f )
```

and

```
x g f ) )
```

respectively. We propose to interpret the expression `x f )` as follows: we shall expect both `x` and `f` to represent words which put an address on the stack `( --- adr)`. The address put on the stack is the code field address of combinator code. The closing parenthesis `)` is simply a synonym for the Forth word `EXECUTE`. When the code associated with `f` gets executed, `x` is taken from the stack and used to compute an address `y` that is left on the stack. Thus, the right parenthesis `)` effectively behaves as a word with the stack picture `(adr1 adr2--adr3)`. The value `y` left on the stack will itself normally be the address of combinator code, possibly a silent word which has been written to the dictionary by the code associated with `f`. We also introduce the words `))` and `)))` for multiple execution. See screen #1 of the accompanying source code.

We introduce a defining word `DEF:` as a convenience for creating functions with the desired stack behavior. A word created by `DEF:` does nothing but put on the stack the address of its associated code. This code is expected to have stack action `(adr1--adr2)`.

For example, if we create a word `INC` by the definition

```
DEF: INC 1 + ;
```

then `INC` by itself only puts an address on the stack, but the code `0 INC )` has exactly the same effect as `0 1+` has. The defining word `DEF:` thus has the effect of creating function words whose action is deferred until `)` is encountered.

A novel feature of combinators, as opposed to ordinary mathematical functions, is that they can be applied to themselves. As a simple example, consider the identity combinator `ID`. (In the literature on combinators, the usual notation is `I`, but in Forth the word `I` already has a meaning; therefore, we use the notation `ID` instead.) The combinator `ID` has the property that it does nothing at all: `x ID )` leaves `x` on the stack. We informally write this as a mathematical equation:

```
x ID ) = x.
```

The equals sign here is not part of the Forth code. This equation makes sense for all `x` and in particular `x` may be replaced by `ID` itself:

```
ID ID ) = ID.
```

Although we have discussed only functions of a single variable, the case of a function of several variables can always be reduced to that of a function of a single variable by a process called *currying* (in honor of Haskell B. Curry, who made this idea more widely known). In mathematics we write $z = f(x,y)$ if z is the value of a function f of two arguments x and y. We write this in S/K/ID as

```
y x f ) ) = z
```

where

```
g = x f )
```

is the function defined by

```
y g ) = z.
```

This behavior is achieved in S/K/ID by having the code `x f )` write a silent word to the dictionary, whose address `g` is left on the stack.

The Primitive Combinators

Only two combinators `K` and `S` are needed to eliminate all variables from equations. The meaning of these primitive combinators must now be explained.

The combinator `K` is used to create constant functions. In our notation the code `y K )` leaves an address on the stack for a silent word

```
: DROP LIT y ;S
```

representing the constant function f with value y. That is,

```
f = y K )
```

satisfies

```
x f ) = y
```

for all x. Eliminating f yields the reduction rule

```
x y K )) = y
```

which is required to hold for all x and y. For example, the constant function

```
K' = ID K )
```

has the value

```
x K' ) = ID
```

for all x.

The similarity of K and K' is revealed by comparing the equations

```
x y K' )) = x
```

and

```
x y K )) = y.
```

We say that K' and K are selectors for the first and second variables, respectively, in these formulas. It is useful to think of the combinator K as a synonym for TRUE and K' as a synonym for FALSE. If this is done, then the expression

```
z y x ))
```

can serve as a combinatory analog of the Forth control structure

```
X IF Y ELSE Z ENDIF.
```

Note that the order of the variables is different. We do not need to introduce any combinator for the conditional control structure at all unless we wish to introduce one solely to change the order of the variables. (See COND in screen #10)

The reduction rule used to define the combinator S is

```
x y z S ))) = x y ) x z )).
```

This equation is required to be valid for all x, y and z. Our definition of S in Forth is motivated by this reduction rule. The code z S) causes a silent word

```
: DEF:, DUP, LIT, ), SWAP, LIT z LIT, )), ;S, ;S
```

to be written to the dictionary, leaving its address on the stack. When this silent word is executed with y on the stack, another silent word

```
: DUP LIT y ) SWAP LIT z )) ;S
```

is written, representing the code y z S)). When this second silent word is executed with x on the stack, finally the code x y) x z)) is executed.

Other combinators can be defined in terms of S and K. For example, Schoenfinkel defines the identity combinator by

```
ID = K K S )).
```

We prefer, however, to introduce `ID` as an additional primitive combinator because its direct definition `DEF: ID ;` produces more efficient code.

Let us use the term *combinatorial expression* to mean either some fixed, definite combinator like `ID`, `K` or `S`, or a variable, like `x` or `y`, or else any expression of the form `a b )` where both `a` and `b` are themselves combinatorial expressions. Any combinatorial expression involving a variable `x` can be rewritten in the form `x f )` where `f` does not involve `x` [LAM80]. The simplest case is

```
x = x ID ).
```

If `c` is an expression which does not involve `x`, we may write

```
c = x c K )) = x f )
```

where

```
f = c K ).
```

For more complicated combinatorial expressions involving `x`, we may find a formula for `f` by using the following inductive procedure. Given any expression of the form `a b )`, where both `a` and `b` may involve `x`, we first write

```
a = x g )
```

and

```
b = x h ),
```

and then

```
a b ) = x g ) x h )) = x g h S ))) = x f )
```

where

```
f = g h S )).
```

As a simple example, the combinatorial expression `x x )` can be written as

```
x x ) = x ID ) x ID )) = x ID ID S ))) = x SELF )
```

where

```
SELF = ID ID S )).
```

The combinators `K'` and `SELF` can be used to give a simple example illustrating the nonassociativity of application. Compare

```
x K' ) SELF ) = ID SELF ) = ID ID ) = ID
```

with

```
x K' SELF )) = x K' K' )) = x ID ) = x.
```

Combinator Equations

It is natural to consider combinators to be equal if they are defined by the same or by equivalent code even though this code may reside at different addresses in memory. Since the addresses of equal combinators need not be the same, one cannot use the Forth word `=` to test for the equality of combinators. Although we lack a convenient Forth word to test for combinator equality, the equality of two given combinators can nevertheless often be deduced by using certain postulated properties of equality [CUR58].

When should two combinators be considered equal? We shall certainly wish to postulate that

if x = y and f = g, then x f) = y g).

We shall also demand that combinator equality be a reflexive, symmetric and transitive relation. That is, we shall demand that f = f for every combinator f, that f = g implies g = f and that if f = g and g = h, then f = h. Finally, we shall postulate an extensionality axiom, considering two combinators to be equal if they produce equal values for every argument. That is,

if x f) = x g) holds for all x, then f = g.

It is precisely this extensional property of equality which makes it possible to eliminate variables from combinator equations. Recall that the primitive combinators S, K and ID may be used to rewrite any expression involving a variable x in the form x f). Therefore, any equation involving a variable x can be written in the form x f) = x g) where f and g do not involve x. If this equation should hold for all x, then we may conclude that f = g.

When several variables occur in an equation, one can eliminate them all one after another. Consider, for example, the reduction rule

x y z B))) = x y) z)

used by Schoenfinkel to define the combinator B. This rule is valid for all x, y and z. One may first eliminate the variable x by rewriting the right-hand side as

x y) x z K))) = x y z K) S))).

Hence, by the extensionality postulate,

y z B)) = y z K) S)).

One may use extensionality again to eliminate the variable y and write

z B) = z K) S).

Finally, to eliminate the variable z, one first rewrites the right-hand side as

z K) z S K))) = z K S K) S)))

and then uses extensionality once more to obtain Schoenfinkel's combinator equation

B = K S K) S)).

Reducing Complexity

Because combinators are determined by their action on arguments, we may define them by algorithms. The complexity of certain standard combinator definitions can be greatly reduced by using procedures instead of equations.

We may assign a crude measure of complexity to combinator definitions as follows: we begin by assigning the complexity 0, 1 and 2, respectively, to the primitive combinators ID, K and S. The complexity of any combinator is defined as the sum of the complexities of the combinators that appear in its definition. For example, by this measure, the procedural definition

```
DEF: B K ) S ) ;
```

for the combinator B has only half the complexity of the standard equational definition

B = K S K) S)).

The combinator B is often used to form composites of functions. The composite h of the functions f and g can be defined by

h = f g B)).

When h is applied to an argument x, one obtains

```
x h ) = x f ) g ),
```

which is the same as one obtains when first f is applied to x and then g is applied to the result. The definition

```
DEF: h f ) g ) ;
```

accomplishes exactly the same thing without using B. Indeed, our definition of B itself reveals it to be the composite of K and S.

Composition satisfies the associative law

```
f g B )) h B )) = f g h B )) B ))
```

for all f, g and h. Indeed,

```
x f g B )) h B ))) = x f g h B )) B )))
                   = x f ) g ) h )
```

holds for all x.

It is convenient to introduce a combinator EVAL satisfying the reduction rule

```
f x EVAL )) = x f )
```

for all x and f. We may interpret the combinator x EVAL) as the process of evaluating at x. To arrive at a definition for EVAL, we rewrite the right-hand side as

```
x f ) = f x K )) f ID ))

      = f x K ) ID S )))
```

and use extensionality to eliminate the variable f, obtaining

```
x EVAL ) = x K ) ID S )).
```

This result reveals the combinator EVAL to be the composite of the two combinators K and ID S).

The combinator W introduced by Curry is used to equate the arguments of a function of two variables. Its reduction rule

```
x f W )) = x x f ))
```

holds for all x and f. Rewriting the right-hand side as

```
x f ) x EVAL )) = x f EVAL S )))
```

and eliminating both x and f using extensionality, we obtain the definition

```
W = EVAL S ).
```

Similar methods may be used to simplify the definition

```
C = K K ) S B B )) S ))
```

given by Schoenfinkel for the combinator C which exchanges the arguments of a function of two variables. Its reduction rule

```
x y f C ))) = y x f ))
```

holds for all x, y and f. First eliminate the variable x to obtain

```
y f C )) = y K ) f S )).
```

Rewrite the right-hand side as

```
y  K  f  S  )  B  )))
```

and eliminate y to obtain

```
f C ) = K f S ) B ))
      = f S ) B ) K EVAL )).
```

We thus arrive at a definition of C as the composite of the three combinators S, B and

```
TAIL = K EVAL ).
```

By our measure, this procedural definition for C has complexity 9 versus 12 for Schoenfinkel's equational definition.

Combinatory Arithmetic

We briefly describe the combinatory interpretation of the arithmetic of the natural numbers. The natural numbers themselves are represented as certain functions. The Church number or iterator corresponding to the number 2 can be defined, for example, as the combinator TWICE satisfying the equation

```
x f TWICE )) = x f ) f )
```

for all x and f. We introduce a successor combinator

```
ENCORE = B S )
```

satisfying

```
f n ENCORE )) = f n B S )))
              = f n ) f B ))
```

for all f and n. We may then count as follows:

```
NEVER = ID K )
ONCE = NEVER ENCORE )
TWICE = ONCE ENCORE )
THRICE = TWICE ENCORE )
   ...
```

The definition of the combinator PLUS corresponding to the arithmetic operation of addition is motivated by a familiar law of exponents as follows: if m and n are iterators, then

```
f m n PLUS ))) = f m ) f n ) B ))
```

expresses the fact that f iterated m n PLUS)) times is the composite of f iterated m times and f iterated n times. The right-hand side here can be rewritten as f m n B B)) S))) and the variables f and m can be eliminated using extensionality, yielding

```
n PLUS ) = n B B )) S ).
```

Hence, the combinator PLUS can be defined as the composite of B B) and S.

The combinator B serves for multiplication. Thus, TWICE THRICE B)) represents two times three. No combinator at all is needed for exponentiation. For example, TWICE THRICE) represents two cubed.

The definition of the predecessor combinator PRED is based on Kleene's original idea. Imagine starting at the origin <0, 0> of a Cartesian plane and taking a walk consisting of n steps, each step going from a point <x,y> to the point <x+1, x>. After n steps we arrive at the point <n,n-1>. We obtain the predecessor n-1 by projecting out the second component. We introduce the combinator START to represent the origin, WALK to represent the steps taken, and TAIL for the final projection. See screen #7.

A zero predicate DONE satisfying

```
NEVER DONE ) = TRUE
```

and

```
n ENCORE ) DONE ) = FALSE
```

for all n can be defined as the composite of FALSE K) EVAL) and TRUE EVAL).

The word INC can be used to test all this arithmetic code. If we write,

```
0 INC TWICE THRICE PLUS )) )).
```

for example, the result 5 will be printed.

List Handling

Lists can be represented in combinatory logic by an application of duality [BAR84, p. 134]. By the dual of x we mean x EVAL). The formula

```
x1 ... xn f ) ... ) = f xn EVAL )) ... x1 EVAL ))
```

suggests that the list

```
x = < x1, ... , xn >
```

may be represented as the reversed composite of the duals of its entries. That is, the combinator representing the list x may be defined as

```
DEF: x xn EVAL )) ... x1 EVAL )) ;
```

If lists are represented this way, then concatenation of lists is (reversed) composition. The identity combinator ID represents the empty list, and the dual of x is the list of the single item x. The individual items in a list can be recovered from this list by replacing f in the previous equation by a suitable selector.

In particular, for the case n=2, if we replace f by the selectors K' and K we find

```
x1 = x1 x2 K' )) = K' x )
   = x K' EVAL )) = x HEAD )
```

and

```
x2 = x1 x2 K )) = K x )
   = x K EVAL )) = x TAIL ).
```

Thus, the combinators HEAD and TAIL project out the first and second coordinates of a list of two items. For example, the combinator START, representing the origin <0, 0>, may be defined as

```
DEF: START NEVER EVAL )) NEVER EVAL )) ;.
```

We do not call it ORIGIN because this word already has a meaning in 64FORTH. The definition in screen #7 is equivalent to this definition because HEAD = NEVER EVAL).

We can now explain some of the details of the definition of the predecessor combinator given in screen #7. If p is the list <x, y> representing some point in the plane, then the point <x+1, x> reached by walking one single step is represented by

```
p WALK ) = x EVAL ) x ENCORE ) EVAL ) B ))
         = x EVAL ) x NEXT ))
         = x EVAL NEXT S )))
         = p HEAD ) EVAL NEXT S )))
```

where we have introduced NEXT, defined as the composite of the three combinators ENCORE, EVAL and B. The point <n, n-1> reached from the origin after n steps is

```
START WALK n )) = n WALK EVAL )) START EVAL ))
```

and therefore

```
n PRED ) = n WALK EVAL )) START EVAL )) TAIL ).
```

The Fixed Point Theorem

Because we have used many combinators to represent rather ordinary functions, it may come as a surprise that every combinator has a fixed point. Before proving this, we introduce some terminology borrowed from Smullyan [SMU85].

We define a combinatory forest to consist of a set F and a mapping) from the Cartesian product set F x F to the set F. That is, for any elements x and f of F, there is an element x f) in F which we call the result of applying f to x.

A forest F is said to be closed under composition if for all f and g in F there exists h in F such that

```
x h ) = x f ) g )
```

holds for all x in F.

An element x is said to be a fixed point for an element f if

```
x f ) = x.
```

Theorem. If a forest F is closed under composition and holds an element SELF satisfying

```
x SELF ) = x x )
```

for all x in F, then every element of the forest has a fixed point.

Proof. Let f be any element, and let g be the composite of SELF and f. Then

```
x g ) = x SELF ) f )
      = x x ) f )
```

holds for all x. Set x equal to g. Hence

```
g g ) = g g ) f )
```

and so g g) is a fixed point of f. Q.E.D.

We have proved more than merely the existence of fixed points; we have explicitly constructed a fixed point. A corollary is that there exists a combinator Y which actually produces a fixed point for any element f of the combinatory forest.

Corollary. Any forest closed under composition and containing ID and S holds an element Y with the property that f Y) is a fixed point of f for every element f.

Proof. Because

```
SELF = ID ID S ))
```

the theorem applies. The combinator

```
OWL = ID S )
```

satisfies

```
x y OWL )) = x y ID S )))
           = x y ) x )
```

for all x and y. By the theorem the combinator OWL has a fixed point

```
Y OWL ) = Y
```

and hence

```
f Y ) = f Y OWL )) = f Y ) f ).
```

Thus f Y) is a fixed point of f. Q.E.D.

The universal existence of fixed points may initially appear to be paradoxical. For example, the combinator ENCORE we have been using for successors of natural numbers has no natural number as a fixed point. The existence of a fixed point for ENCORE is no real paradox, however, because ENCORE can be applied to anything, not just to numbers. Similarly, the logical combinator NOT can have neither TRUE nor FALSE as fixed points, but there is no contradiction because we can apply NOT to objects other than TRUE and FALSE.

Delaying Execution

It should not be surprising that it is possible to write combinatory code which causes the computer to crash. In this respect the combinatory programming language S/K/ID is no different from any other language in which loops can be written.

The simplest example of code that causes a crash is

```
SELF SELF ) = SELF ID ID S )))
            = SELF ID ) SELF ID )) = SELF SELF ID ))
            = SELF SELF ) = ...
```

The computer goes into an endless loop, causing the dictionary to explode as more and more silent words are compiled. A rather similar crash occurs when the code W W W)) is executed.

A more interesting crash is caused by the paradoxical combinator Y defined by

```
DEF: Y SELF B' )) SELF ) ;.
```

The combinator B' here is defined by

```
DEF: B' K ) ENCORE ) ;
```

and satisfies the reduction rule

```
x f g B' ))) = x g ) f )
```

for all x, f and g. The paradoxical combinator Y as defined here will always crash because

```
f Y ) = f SELF B' )) SELF )
      = f SELF B' )) f SELF B' )))
      = f SELF B' )) SELF ) f )
      = f Y ) f )
      = f Y ) f ) f )
      = ...
```

Because words in Forth are executed from left to right, in the expression `x y ) z )`, the code for `y` must execute before the code for `z` can execute. It follows, for example, that in the expression `SELF SELF ) K' )` the computer will crash before the code associated with `K'` will get a chance to execute. If we wish to avoid this crash, we must somehow arrange for the execution of `y` to be delayed [HEN76].

One way to postpone execution is by introducing a new word for this very purpose. The delayed execution word `.)` is defined so that `x y .)` does not cause `y` to be executed immediately, but instead causes a silent word to be written to the dictionary leaving a memorandum for the execution of `y`. The address of this silent word is left on the stack. The new silent word will be designed so that

```
x y z .) ) = x y z ))
```

holds for all `x`, `y` and `z`.

The expression `SELF SELF .) K' )` will not cause a computer crash but will cause `K'` to execute first, leaving the result `ID` on the stack.

The crash caused by the paradoxical combinator `Y` can be avoided by replacing the executor `)` with the delayed executor `.)` in some strategic places. If `Y` can be redefined so that

```
f Y ) = f Y .) f )
```

then we could write, for example,

```
K' Y ) = K' Y .) K' ) = ID
```

without a crash occurring.

The necessary delaying effect can be achieved by building an explicit delay into the first `SELF` occurring in the definition of the paradoxical combinator. This can be done with the delayed executor as shown in screen #13 of the accompanying source code.

It is not strictly necessary to introduce the delayed executor to achieve a delay effect. Without adding any new primitives to S/K/ID, we can produce delays in execution by using double transposition. Define a combinator `DELAY` as the composite of `C` with itself. Its reduction rule is

```
x y z DELAY ))) = x y z )).
```

The combinator `DELAY` behaves much like `ID` but with a delayed action.

Replacing the first `SELF` with `SELF DELAY )` achieves the same effect as using a delayed executor. Because the combinator `C` is fairly complicated in pure S/K/ID, a more efficient procedure is to define the double transpose of `SELF` directly as shown in screen #11.

Using delayed execution, the paradoxical combinator serves a useful purpose for implementing recursion [BUR75]. We may think of `Y` perhaps as a combinatory analog for the Forth control structure `BEGIN ... AGAIN`.

Recursive Definition of Factorial

To test our proposals for using delays in implementing the paradoxical combinator, it may be useful to consider a specific application to the recursive algorithm for computing the factorial function.

Our starting point is the recursion relation which defines the factorial function `FACT`:

```
n FACT ) = n PRED ) FACT ) n B )) ID n DONE ))).
```

This says that `n` factorial is equal to one if `n` is zero and otherwise is equal to `n-1` factorial times `n`.

We first write

```
n PRED ) FACT ) = n FACT PRED B' )))
```

and replace

```
ID n DONE ))
```

by the equivalent code

```
n START ) = K' K' n ))
```

to obtain

```
n FACT ) = n FACT PRED B' ))) n B )) n START ))
         = n FACT PRED B' )) B S ))) n START ))
         = n FACT PRED B' )) ENCORE ) START S ))).
```

From this we can eliminate n to obtain

```
FACT = FACT PRED B' )) ENCORE ) START S )).
```

which says that FACT is a fixed point of the combinator ACT defined by

```
DEF: ACT PRED B' )) ENCORE ) START S )) ;.
```

That is,

```
FACT = FACT ACT )
```

and therefore FACT may be defined using the paradoxical combinator Y as

```
FACT = ACT Y ).
```

To test this, we might try

```
Ø INC THRICE FACT ))).
```

which should yield 6. When we do this, however, the computer crashes because we have failed to build in delays.

We need to build delays not only in the paradoxical combinator Y itself, as discussed in the preceding section, but also in the combinator ACT to which it is applied. This is important because the termination condition for the loop needs to be checked first. As shown in screen #12, this can be accomplished simply by inserting a DELAY into the definition of ACT.

With these delays built in, the factorial combinator does work. A simple test shown in screen #15 correctly computes (2*3)!=720. The test also reveals that the silent words generated by this computation occupy 17828 bytes of memory. Lack of efficient garbage collection techniques prevents the computation of the factorial of larger numbers.

Additional Primitives

Being able to base a language for functions on only the two primitive combinators S and K is very attractive from a theoretical point of view. From a practical standpoint, however, introducing more primitives may significantly reduce the amount of dictionary space devoted to silent words generated during a computation. Many people have explored the advantages of additional or alternative primitives. Curry, for example, preferred to use ID, K, B, C and W as primitives. Another set of primitives is ID, K', SELF, EVAL and B.

The simplest of these combinators to implement directly in Forth as new primitives are K' and SELF. These can be defined by

```
DEF: K' DROP ID ;
```

and

```
DEF: SELF DUP ) ;
```

respectively.

Next in order of complexity are W and EVAL. These can be defined directly by

```
DEF: W DEF:, DUP, LIT, )), ;S, ;
```

and

```
DEF: EVAL DEF:, LIT, SWAP, ), ;S, ;
```

respectively.

Direct definitions of B and B' are given by

```
DEF: B DEF:, DEF:,, LIT,, ),,
       LIT, LIT,, ),, ;S,, ;S, ;
```

and

```
DEF: B' DEF:, DEF:,, LIT, LIT,, ),,
        LIT,, ),, ;S,, ;S, ;
```

while C may be directly defined by

```
DEF: C DEF:, DEF:,, LIT,, SWAP,,
       LIT, LIT,, )),, ;S,, ;S, ;.
```

If desired, even more primitives can be added.

We have already discussed the possibility of introducing a delayed action execution word. Both) and .) have the same stack action (a1 a2--a3). Other words with this stack action can be introduced. For example, it may be useful to introduce a composition word B)) by the definition

For example, it may be useful to introduce a composition word B)) by the definition

```
: B)) HERE SWAP ROT :, LIT, ), LIT, ), ;S, ;
```

so that one can write x y B)) instead of x y B)) for the composite of x and y. The advantage here is that only one silent word is written, instead of two. One can introduce a reversed composition word B')) by

```
: B')) SWAP B)) ;.
```

For list processing, the word EVAL)) defined by

```
: EVAL)) SWAP ) ;
```

is useful. This has the property

```
f x EVAL)) = x f )
```

making EVAL)) equivalent to EVAL)) but causing fewer silent words to be written. An obvious application is to further simplify the definition of the predecessor combinator given in screen #7.

An even more interesting possibility is to use the word

```
: MYSELF LATEST PFA CFA , ; IMMEDIATE
```

to define primitive combinators which are fixed points of some standard combinators. The simplest possibility here is the combinator U defined by

```
DEF: U DROP MYSELF ;.
```

This combinator has the property

```
x U ) = U
```

which suggests interpreting U as "undefined." Since

```
x U ) = U = x U K ))
```

we can eliminate x, revealing

```
U = U K ).
```

That is, the combinator U is a fixed point of the combinator K. Another curious example along these lines is the mysterious combinator X defined by

```
DEF: X MYSELF SWAP ) ;.
```

This combinator is a fixed point of EVAL and has the strange property

```
a X ) = X a ).
```

The practical value of such a combinator is unclear.

A more practical application of this recursive technique is the following definition of the delayed paradoxical combinator

```
DEF: Y. DUP MYSELF .) SWAP ) ;.
```

Compiling Combinator Code

Programming directly in terms of combinators may yield efficient code, but it is often more convenient to write in a higher order language like the lambda calculus and then use a compiler to make the necessary translations.

In a previous section we discussed how to systematically create a combinator satisfying any given reduction rule by using Schoenfinkel's recursive method for eliminating variables. For this only the combinators ID, K and S were needed. The process is tedious to do by hand, but we can, of course, do it by computer [BUR75, p. 42]. Any such procedure can be considered as an algorithm for compiling combinator code.

Different algorithms for compiling combinator code are possible, and some lead to simpler code than others ([MEI84]; [HIR85]; [NOS85]). When additional primitives are available, new possibilities for optimization arise.

In particular, if B, C and S are all available as primitives, then the special rules

```
x L ) R ) = x L R B )) )
```

and

```
L x R ) ) = x L R C )) )
```

can be used in addition to the rule

```
x L ) x R ) ) = x L R S )) )
```

mentioned in *Combinator Equations*. The order in which these rules are applied can affect the results obtained [CUR72, pp. 43ff].

Consider, for example, the problem of constructing compositor combinators for all the different ways that parentheses can be inserted into an expression [SMU85, pp. 95-98]. There are only two ways to do this for three variables, namely,

```
a b c ) ) = a b c ID )))
a b ) c ) = a b c B ))).
```

For four variables, there are five ways to insert parentheses, corresponding to five compositor combinators:

```
a b c d )))    = a b c d ID          )) ))
a b c ) d ))   = a b c d B           )) ))
```

```
a b ) c d ))    = a b c d B B )          )) ))
a b c )) d )    = a b c d B B B ))       )) ))
a b ) c ) d )   = a b c d B B B ) B ))   )) ))
```

We can generalize these results to any number of variables, and it turns out that all we ever need to construct these compositor combinators are `ID` and `B`. It is not even hard to write a formal algorithm which constructs all such combinators.

In the first four cases, the method of eliminating variables immediately yields the results given above. But consider what happens if we blindly follow the elimination method for the fifth case. Following Smullyan, we call the required combinator `BECARD`:

```
a b c d BECARD )) )) = a b ) c ) d ).
```

By a direct application of the above rules, eliminating each of the variables `a`, `b`, `c` and `d` in turn, without any further simplifications, yields successively

```
b c d BECARD )))      = b c B )) d B ))
c d BECARD ))         = c B ) d B ) B ))
d BECARD )            = B d B ) B ) B ))
BECARD                = B B B B )) B B )) C )).
```

This final result is correct albeit rather complicated. The algorithm works, but we can do better. To obtain a simpler result, we first use the definition of `B` to rewrite our original expression as follows:

```
a b ) c ) d )   = a b ) c d B )))
                = a b c d B )) B ))).
```

We now eliminate `a` and `b` and again rewrite the resulting expression

```
c d BECARD ))   = c d B )) B )
                = c d B ) B B ))).
```

Eliminate `c` next and rewrite once more to obtain

```
d BECARD )      = d B ) B B ))
                = d B B B ) B ))).
```

Finally, we eliminate `d` and find the simple result

```
BECARD = B B B ) B )).
```

The same rules were applied, with the only difference being the order in which they were applied.

Outlook

We have presented a tiny language S/K/ID to show how a combinator-based functional programming language can be written in Forth. Some ways to simplify this language and make it more practical were suggested.

Using `FORGET` to collect garbage only at the end of computations will not suffice for more complex programs. One can reduce the number of silent words produced by shifting work to the data stack from the dictionary as explained in *Additional Primitives*. Nevertheless, better methods of garbage collection should be explored ([SCH67]; [WIL84]; [CHR84]; [LI86]; [GLA87]). Because delays generate extra code, they should be avoided unless they are essential. Careful analysis is required to determine precisely where to use delays to prevent crashes ([HEN76]; [TUR85]; [CLA85]).

Finally, it may be interesting to explore whether any techniques suggested here can find use in existing implementations of functional programming languages in Forth which do not use combinators at all [DIX84].

References

[ABD76] Abdali, S. K. 1976. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic* 41:222-24.

[BAC78] Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21:613-41.

[BAR84] Barendregt, H. P. 1984. *The lambda calculus—its syntax and semantics*. Rev. ed. Amsterdam: North-Holland Publishing Co.

[BAR80] Barwise, J., Keisler, H. J., and Kunen, K., eds. 1980. *The Kleene symposium.* Proceedings of the symposium held June 18-24, 1978, at Madison, Wisconsin, U.S.A. Amsterdam: North-Holland Publishing Co.

[BOE82] Boehm, C. 1982. Combinatory foundations of functional programming. *Conference record of the 1982 ACM Symposium on LISP and functional programming*. Pittsburgh, PA: ACM.

[BUR75] Burge, W. H. 1975. *Recursive programming techniques*. Reading, MA: Addison-Wesley Publishing Co.

[CHR84] Christopher, T. W. 1984. Reference count garbage collection. *Software Practice and Experience* 14:503-07.

[CHU33] Church, A. 1933. A set of postulates for the foundation of logic. *Ann. of Math.* 34:839-64.

[CHU41] Church, A. 1941. *The calculi of lambda conversion*. 2nd ed., 1951. Princeton, NJ: Princeton University Press.

[CLA85] Clack, C., and Peyton Jones, S. L. 1985. Strictness analysis – A practical approach. In Jouannaud, J. P., ed. *Functional programming languages and computer architecture. Lecture notes in computer science.* Vol. 201, pp. 35-49. New York: Springer-Verlag.

[CLA80] Clarke, T. J. W., Gladstone, P. J. S., Maclean, C. D., and Norman, A. C. 1980. SKIM – the S, K, I reduction machine. *ACM conference on LISP and functional programming languages*, pp. 128-35. ACM.

[CUR30] Curry, H. B. 1930. Foundations of combinatory logic [in German]. *Amer. J. Math.* 52:509-36;789-834.

[CUR58] Curry, H. B., and Feys, R. 1958. *Combinatory logic, volume 1.* Amsterdam: North-Holland Publishing Co.

[CUR72] Curry, H. B., Hindley, J. R., and Seldin, J. P. 1972. *Combinatory logic, volume 2.* Amsterdam: North-Holland Publishing Co.

[DIX84] Dixon, R. D., Edmondson, W. M., Franklin, R. D., and Sloan, J. L. 1984. Extensions of FORTH for functional programming. *1984 Rochester Forth conference*, pp. 228-33. Rochester, NY: Institute for Applied Forth Research, Inc.

[EIS85] Eisenbach, S., and Sadler, C. 1985. Declarative languages: an overview. *BYTE* 10(8):181-97.

[GLA84] Glaser, H., Hankin, C., and Till, D. 1984. *Principles of functional programming*. Englewood Cliffs, NJ: Prentice-Hall International.

[GLA87] Glaser, H. W., and Thompson, P. 1987. Lazy garbage collection. *Software Practice and Experience* 17:1-4.

[HEN76] Henderson, P., and Morris, J. H. 1976. A lazy evaluator. *Third symposium on principles of programming languages*, pp. 95-103. Atlanta, GA: ACM.

[HIK84] Hikita, T. 1984. On the average size of Turner's translation to combinator programs. *Journal of Information Processing* 7:164-69.

[HIN86] Hindley, J. R., and Seldin, J. P. 1986. *Introduction to combinators and lambda-calculus*. Cambridge, England: Cambridge University Press.

[HIR85] Hirokawa, S. 1985. Complexity of the combinator reduction machine. *Theoretical Computer Science* 41:289-303.

[HUD84] Hudak, P., and Kranz, D. 1984. A combinator-based compiler for a functional language. *Conference record of the 11th annual symposium on principles of programming languages*, pp. 121-32. Salt Lake City, UT.

[HUG82] Hughes, R. J. M. 1982. Supercombinators—a new implementation method for applicative languages. *Conference record of the 1982 ACM symposium on LISP and functional programming*, pp. 1-10. Pittsburgh, PA: ACM.

[JOH84] Johnson, T. 1984. Efficient compilation of lazy evaluation. *ACM SIGPLAN symposium on compiler construction. SIGPLAN Notices* 19(6): 58-69.

[JOU85] Jouannaud, J. P., ed. 1985. *Functional programming languages and computer architecture. Lecture notes in computer science.* Vol. 201. New York: Springer-Verlag.

[KLE35] Kleene, S. C., and Rosser, J. B. 1935. The inconsistency of certain formal logics. *Ann. Math.* 36:630-36.

[KLE36] Kleene, S. C. 1936. Lambda-definability and recursiveness. *Duke Mathematical Journal* 2:340-53.

[KLE80] Kleene, S. C. 1980. Recursive functionals and quantifiers of finite types revisited II. In Barwise, J., Keisler, H. J. and Kunen, K., eds. *The Kleene symposium*. Proceedings of the symposium held June 18-24, 1978, at Madison, Wisconsin, U.S.A., pp. 1-29. Amsterdam: North-Holland Publishing Co.

[KLE81] Kleene, S. C. 1981. The theory of recursive functions: Approaching its centennial. *Bulletin of the Amer. Math. Soc.* 5:43-61.

[LAM80] Lambek, J. 1980. From lambda calculus to Cartesian closed categories. In Seldin, J. P., and Hindley, J. R., eds. *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pp. 375-402. New York: Academic Press.

[LAM86] Lambek, J., and Scott, P. J. 1986. *Introduction to higher order categorical logic*. Cambridge, England: Cambridge University Press.

[LAN64] Landin, P. J. 1964. The mechanical evaluation of expressions. *Computer Journal* 6:308-20.

[LI86] Li, K., and Hudak, P. 1986. A new list compaction method. *Software Practice and Experience* 12:145-63.

[MAU83] Maurer, P. M., and Oldehoeft, A. E. 1983. The use of combinators in translating a purely functional language to low-level data-flow graphs. *Computer Languages* 8:27-45.

[MEI84] Meira, S. L. 1984. Optimised combinatoric code for applicative language implementation. *Proc. of the 6th intern. symposium on programming*. New York: Springer-Verlag.

[MEN87] Mendelson, E. 1987. *Introduction to mathematical logic*. 3rd ed. Belmont, CA: Wadworth & Brooks/Cole.

[NOS85] Noshita, K., and Hikita, T. 1985. The BC-chain method for representing combinators in linear space. In *RIMS symposia on software science and engineering II*, pp. 292-306. *Lecture Notes in Computer Science*. Vol. 220. New York: Springer-Verlag.

[PET81] Peter, R. 1981. *Recursive functions in computer theory*. [originally published in German, 1976; translated into English by I. Juhasz.] New York: Halsted Press, John Wiley and Sons.

[RUS72] Rustin, R., ed. 1972. *Formal semantics of programming languages*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[SCH24] Schoenfinkel, M. 1924. On the building stones of mathematical logic. *Mathematische Annalen* 92:305-16 [in German]. English translation in J. van Heijenoort. 1967. *From Frege to Goedel: A source book in mathematical logic, 1879-1931*, pp. 355-66, Cambridge, MA: Harvard University Press.

[SCH67] Schorr, H., and Waite, W. 1967. An efficient machine-independent procedure for garbage collection of various list structures. *Communications of the ACM* 10:501-06.

[SCO72] Scott, D. 1972. Lattice theory, data types and semantics. In Rustin, R., ed. *Formal semantics of programming languages*, pp. 65-106. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[SCO77] Scott, D. 1977. Logic and programming languages. *Communications of the ACM* 20:634-41.

[SCO80] Scott, D. 1980. Lambda calculus: Some models, some philosophy. In Barwise, J., Keisler, H. J., and Kunen, K., eds. *The Kleene symposium*. Proceedings of the symposium held June 18-24, 1978, at Madison, Wisconsin, U.S.A., pp. 223-65. Amsterdam: North-Holland Publishing Co.

[SEL80] Seldin, J. P., and Hindley, J. R., eds. 1980. *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*. New York: Academic Press.

[SHA85] Sharp, J. A. 1985. *Data flow computing*. New York: Halsted Press, John Wiley and Sons.

[SMU85] Smullyan, R. 1985. *To mock a mockingbird, and other logic puzzles*. New York: Alfred A. Knopf.

[STE72] Stenlund, S. 1972. *Combinators, lambda-terms and proof theory*. Dordrecht, Holland: D. Reidel Publishing Co.

[STO77] Stoy, J. E. 1977. *Denotational semantics: The Scott-Strachey approach to programming language theory*. Cambridge, MA: MIT Press.

[TUR79a] Turner, D. A. 1979. A new implementation technique for applicative languages. *Software Practice and Experience* 9:31-49.

[TUR79b] Turner, D. A. 1979. Another algorithm for bracket abstraction. *Journal of Symbolic Logic* 44:267-70.

[TUR85] Turner, D. A. 1985. Functional programs as executable specifications. In *Mathematical Logic and Programming Languages*, edited by C. A. R. Hoare and J. C. Shepherdson, pp. 29-54. Englewood Cliffs, NJ: Prentice-Hall International.

[WIL84] Wilson, J. 1984. FORTH execution on non-von Neumann machines. In *1983 FORML conference*, pp. 15-28. San Jose, CA: Forth Interest Group.

Manuscript received February 1987.

Johan G. F. Belinfante holds a B.S. from Purdue University and a Ph.D. from Princeton University in physics. Following postdoctoral studies in elementary particle physics at the California Institute of Technology and a position as research investigator at the University of Pennsylvania, he was an assistant professor in physics and an associate professor in physics and mathematics at Carnegie-Mellon University. Currently he is a professor in applied mathematics at the Georgia Institute of Technology. He has long been active in the applications of computers to abstract algebra, including Lie algebras, group theory and lattice theory and is a co-author of a book on Lie algebras. He has programming experience in many computer languages and has published programs in ALGOL and FORTRAN.

```
SCR # 1
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 1 OF 15              )
 2
 3 ( COMPILE DOCOLON CODE ADDRESS)
 4 : :,   LIT [COMPILE] : @ , ;
 5
 6 ( DEFINOR FOR DEFERRED-ACTION WORDS)
 7 : DEF: [COMPILE] : 2 ALLOT :, DOES> ;
 8
 9 ( EXECUTOR)
10 : )    EXECUTE ;
11
12 ( MULTIPLE EXECUTION)
13 : ))   ) ) ;
14 : )))  ) ) ) ;
15

SCR # 2
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 2 OF 15              )
 2
 3 ( SOME COMPILING WORDS)
 4
 5 : DEF:,   HERE SWAP :, ;
 6 : LIT,    LIT LIT , ,  ;
 7 : ),      LIT )     ,  ;
 8 : )),     LIT ))    ,  ;
 9 : DROP,   LIT DROP  ,  ;
10 : DUP,    LIT DUP   ,  ;
11 : SWAP,   LIT SWAP  ,  ;
12 : ;S,     LIT ;S    ,  ;
13
14
15

SCR # 3
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 3 OF 15              )
 2
 3 ( HIGHER ORDER COMPILING WORDS)
 4
 5 : DEF:,,  LIT DEF:, , ;
 6 : LIT,,   LIT LIT,  , ;
 7 : ),,     LIT ),    , ;
 8 : )),,    LIT )),   , ;
 9 : DUP,,   LIT DUP,  , ;
10 : SWAP,,  LIT SWAP, , ;
11 : ;S,,    LIT ;S,   , ;
12
13
14
15
```

```
SCR # 4
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 4 OF 15              )
 2
 3 ( IDENTITY COMBINATOR)
 4 DEF:  ID   ;
 5
 6 ( SCHOENFINKEL'S CONSTANT COMBINATOR K)
 7 DEF:  K    DEF:,  DROP, LIT, ;S, ;
 8
 9 DEF:  K'   ID K )) ;
10
11 ( SCHOENFINKEL'S SMELTOR COMBINATOR S)
12 DEF:  S    DEF:, DEF:,, DUP,, LIT,, ),,
13            SWAP,, LIT,  LIT,,  )),,
14            ;S,,  ;S,  ;
15

SCR # 5
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 5 OF 15              )
 2 ( SMULLYAN'S OWL)
 3 DEF: OWL    ID S )) ;
 4
 5 ( SELF-APPLICATOR)
 6 DEF: SELF   ID OWL )) ;
 7
 8 ( COMPOSITION COMBINATOR)
 9 DEF: B      K ) S ) ;
10
11 ( EVALUATION COMBINATOR)
12 DEF: EVAL   K ) OWL ) ;
13
14 ( IDENTIFY ARGUMENTS)
15 DEF: W      EVAL S )) ;

SCR # 6
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 6 OF 15              )
 2
 3 ( SOME LOGICAL COMBINATORS)
 4
 5 DEF: TRUE    K  ) ;
 6 DEF: FALSE   K' ) ;
 7
 8 DEF: TAIL    TRUE  EVAL )) ;
 9 DEF: HEAD    FALSE EVAL )) ;
10
11 ( INCLUSIVE-OR COMBINATOR)
12 DEF: V       SELF ) ;
13 DEF: &       K  S  S ))) ;
14 DEF: NOT     HEAD ) TAIL ) ;
15
```

```
SCR # 7
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 7 OF 15              )
 2
 3 ( SUCCESSOR COMBINATOR)
 4 DEF: ENCORE   B S )) ;
 5
 6 ( COMBINATORY ADDITION)
 7 DEF: PLUS     B B )) S ) ;
 8
 9 DEF: START    HEAD ) HEAD ) ;
10 DEF: NEXT     ENCORE ) EVAL ) B ) ;
11 DEF: WALK     HEAD ) EVAL NEXT S ))) ;
12
13 ( PREDECESSOR COMBINATOR)
14 DEF: PRED     WALK  EVAL ))
15               START EVAL )) TAIL ) ;

SCR # 8
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 8 OF 15              )
 2
 3 ( ZERO PREDICATE)
 4 DEF: DONE   FALSE K ) EVAL )) TAIL ) ;
 5
 6 ( CHURCH NUMBERS OR ITERATORS)
 7 DEF: NEVER  K' ) ;
 8
 9 DEF: ONCE   ID ) ;
10
11 DEF: TWICE  ONCE  ENCORE )) ;
12
13 DEF: THRICE TWICE ENCORE )) ;
14
15

SCR # 9
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 9 OF 15              )
 2 ( REVERSED COMPOSITION)
 3 DEF:  B'    K ) ENCORE )   ;
 4
 5 ( ROTATION)
 6 DEF:  RHO   EVAL ) B )     ;
 7
 8 DEF:  PHI   B ) S B ))     ;
 9
10 ( SWAP ARGUMENTS)
11 DEF:  C     S ) B ) TAIL ) ;
12
13 ( SCOTT'S PAIRING)
14 DEF:  DYAD  EVAL ) C )     ;
15
```

```
SCR # 10
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 10 OF 15              )
 2
 3 ( IMPLICATION)
 4 DEF: IMP     HEAD ) NOT ) ;
 5
 6 ( CHURCH PAIRING)
 7 DEF: PAIR    K ) DYAD ) ;
 8
 9 ( CONDITIONAL)
10 DEF: COND    RHO ) C ) ;
11
12 DEF: PSI     B B )) B ) RHO S )) ;
13
14
15

SCR # 11
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 11 OF 15              )
 2
 3 DEF: DELAY    C ) C ) ;
 4
 5 ( TRANSPOSED SELF)
 6 DEF: SELF'    K ) SELF  S )) ;
 7
 8 ( DOUBLE TRANSPOSED SELF)
 9 DEF: SELF''   K ) SELF' S )) ;
10
11 ( DELAYED PARADOXICAL COMBINATOR)
12 DEF: Y''      SELF'' B' )) SELF ) ;
13
14
15

SCR # 12
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (          SCREEN 12 OF 15              )
 2
 3 ( EXAMPLE USING PARADOXICAL COMBINATOR)
 4
 5 DEF: ACT      PRED B' )) ENCORE )
 6               DELAY ) START S )) ;
 7
 8 ( FACTORIAL)
 9 DEF: FACT''   ACT Y'' )) ;
10
11
12
13
14
15
```

```
SCR # 13
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (         SCREEN 13 OF 15              )
 2
 3 ( DELAYED EXECUTOR)
 4 : .)        HERE SWAP ROT
 5             :, LIT, LIT, )), ;S, ;
 6
 7 ( DELAYED SELF)
 8 DEF: SELF.  SELF .) ;
 9
10 ( DELAYED PARADOXICAL COMBINATOR)
11 DEF: Y.     SELF.  B' )) SELF ) ;
12
13 ( FACTORIAL)
14 DEF: FACT.  ACT Y.  )) ;
15

SCR # 14
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (         SCREEN 14 OF 15              )
 2 DEF: INC 1+ ;
 3
 4 : ARITHMETIC ;  CR HERE H. CR DECIMAL
 5 CR SP! 0 INC NEVER  )) .S
 6 CR SP! 0 INC ONCE   )) .S
 7 CR SP! 0 INC TWICE  )) .S
 8 CR SP! 0 INC THRICE )) .S
 9 CR SP! 0 INC TWICE THRICE PLUS )) )) .S
10 CR SP! 0 INC TWICE THRICE B    )) )) .S
11 CR SP! 0 INC TWICE THRICE  ))) .S
12 CR SP! 0 INC THRICE TWICE  ))) .S
13 CR SP! 0 INC THRICE PRED   ))) .S
14 CR SP! HERE H. CR FORGET ARITHMETIC
15

SCR # 15
 0 ( FILE: S/K/ID.REVISED    1987 MAR. 24)
 1 (         SCREEN 15 OF 15              )
 2
 3 : ARITHMETIC ;  CR HERE H. CR DECIMAL
 4 DEF: 4X    TWICE TWICE )) ;
 5 DEF: 16X   TWICE 4X ))    ;
 6 DEF: 256X  4X 4X ))  ;
 7 SP! 0 INC 256X ) 16X ) TWICE )) .
 8 CR SP! HERE H. CR FORGET ARITHMETIC
 9
10 : ARITHMETIC ;  CR HERE H. CR DECIMAL
11 SP! 0 INC TWICE THRICE B )) FACT'' ))) .
12 CR SP! HERE H. CR FORGET ARITHMETIC
13
14
15
```