Preliminary Investigation of Random SKI-Combinator Trees

Bruce J. MacLennan*

Computer Science Department
University of Tennessee, Knoxville
maclennan@cs.utk.edu

October 20, 1997

Abstract

SKI-combinator trees are a simple model of computation, which are computationally complete (in a Turing sense), but are suggestive of basic biochemical processes and can be used as a vehicle for understanding processes of biological (and prebiotic) self-organization. After a brief overview of SKI-combinator trees, we describe the results of a series of preliminary experiments exploring the statistical properties of populations of random SKI-combinator trees. We show that in such populations a significant fraction of the trees will exhibit complex, non-terminating growth patterns, suggestive of biological processes. Further, we show that the fraction of S-combinators in such trees is an important parameter defining a sharp phase transition between (uninteresting) terminating behavior and (interesting) nonterminating growth. (This is related to the "edge of chaos" investigated by Chris Langton.) Finally, we discuss some of the follow-on investigations suggested by these exploratory experiments.

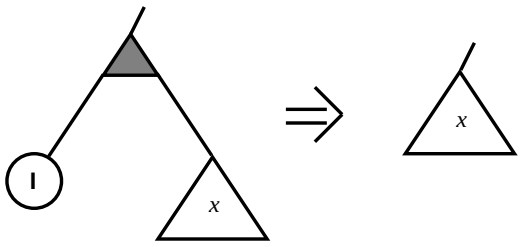
*Most of the research reported herein was conducted while the author was a Fellow of the Institute for Advanced Studies of the Collegium Budapest, June–July 1997, whose support is gratefully acknowledged.

1 Introduction

The immediate motivation for this project was two papers (Szathmáry 1995, 1997) that criticize a certain lambda-calculus model of biochemical organization (Fontana & Buss 1994). It occurred to me that a different and much simpler formal system, known as *combinatory logic* (specifically, the SKI *calculus*), would be a simpler and more natural model of biochemical processes.[1] Therefore I have conducted a series of exploratory computer experiments investigating the properties of "SKI soup," that is, random structures (binary trees) made from the three atoms S, K and I. This report presents the results of these preliminary investigations, which reveal interesting self-organizing behavior and suggest many follow-on experiments.

1.1 Summary of SKI Combinators

I will begin with a brief, informal description of the SKI calculus.[2] The I combinator, called the *elementary identificator*, is the identity function, which is defined by the rewrite rule $\mathsf{I}x \Rightarrow x$. This can be seen as a tree-rewrite:

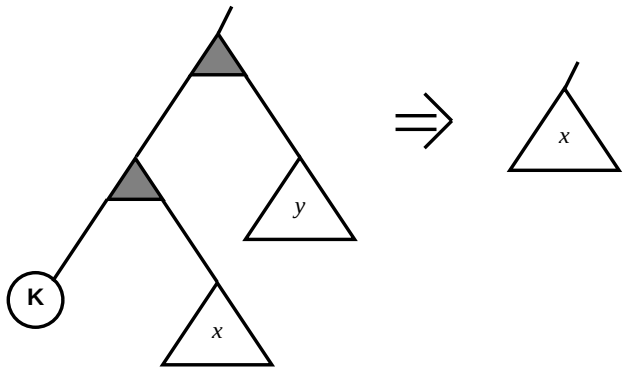


Thus, the only effect of I on the tree is to delete itself; it is effectively a "do nothing" operation.

The K combinator, called the *elementary cancellator*, makes a constant function, which is defined by the rewrite rule $\mathsf{K}xy \Rightarrow x$, or by the tree-rewrite:

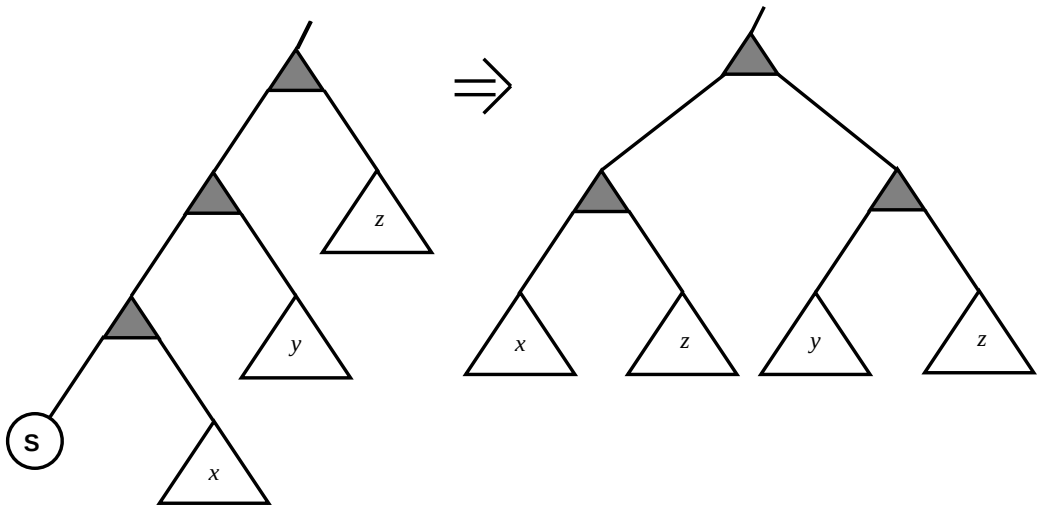
[1] The primary reference for combinatory logic is Curry & al. (1958); Barendregt (1984, esp. chs. 7, 9) also has a comprehensive discussion. Less formal presentations can often be found in books on *functional programming* (e.g. MacLennan 1990).

[2] The SKI calculus is described briefly in MacLennan (1990, pp. 442–445) and in many other books on functional programming; the definitive description is Curry & al. (1958).

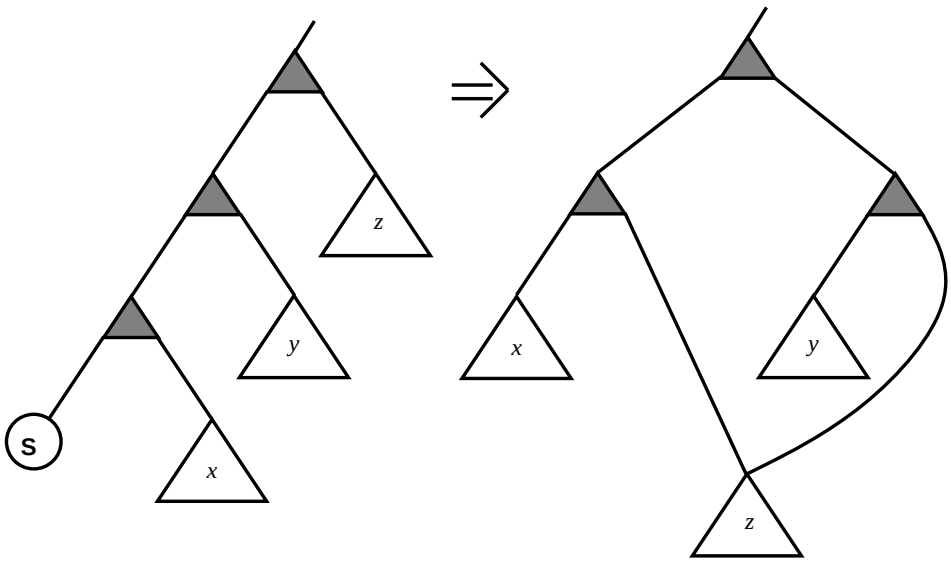


Thus, K discards information (specifically, y in Kxy), which could be a whole subtree; in effect it breaks the link to y. K can also be thought of as a projection operation, since Kxy replaces the pair xy by its first component x.

The S combinator, which may be called *lifted application* (MacLennan 1990, p. 442), is difficult to describe in words, but the operation it performs is simple. The rewrite rule is S$xyz \Rightarrow xz(yz)$. Since the convention is that in the absence of parenthesis, symbols associate to the left, the result of S is equivalent to $((xz)(yz))$. In tree form:



This can be thought of in two ways: either the tree z is duplicated (as depicted above), or two links are created to the single instance of z:



It makes no difference as far as combinatory logic is concerned; both interpretations give equivalent trees.

However, they have differing chemical interpretations. Interpreting S as duplicating its third argument presupposes some lower level chemical replication processes (capable of duplicating a binary tree). Such processes are known, but they are not simple. On the other hand, if we interpret S as creating an additional link to its third argument, and if we interpret the links as chemical bonds, then we have the chemically unrealistic situation of an unlimited number of bonds to some structure. Deciding which interpretation is most plausible may depend on which specific chemical processes the SKI combinators are intended to model.

1.2 Why SKI?

There are several reasons for investigating SKI-combinator trees:

1. It is a minimal computationally complete (in a Turing sense) set.[3] That is, expressions formed from S, K and I are able to compute any function computable on a Turing machine or on any finite digital computer.

[3] Although SKI is commonly taken as a minimal set of combinators, I is definable in terms of the other two, $\mathsf{I} \equiv \mathsf{SKK}$ or SKS, so it would be interesting to investigate SK-trees as models of biochemical organization.

(There are, however, other computationally complete sets of combinators, and it will be worthwhile to investigate them, since they might be more biochemically plausible.[4])

2. The SKI combination has been extensively studied, and even investigated as a basis for practical digital computers (e.g. Fasel & Keller, 1987; Turner 1979).

3. The functions of these three combinators are nearly independent:

   - I has very little effect on the tree, since it replaces itself by its argument.
   - K deletes its second argument, and thus can delete an entire subtree.
   - S duplicates its third argument, and thus can duplicate an entire subtree.

   There are other sets of combinators having these three functions but are not computationally universal, and it will be worthwhile to see if they lead to the same kinds of behavior.[5]

1.3 Normal Order

Combinatory logic, like the lambda-calculus, satisfies the Church-Rosser Property, which means that all orders of applying the rules to a tree give the same result, if they give a result at all.[6] That is, all terminating computations give

[4]Other computationally complete sets of combinators include: (1) B′, K and W, where $\mathsf{B}'xyz \Rightarrow y(xz)$ (*reverse compositor*) and $\mathsf{W}xy \Rightarrow xyy$ (*elementary duplicator*); (2) B, K, C_*, W_*, where $\mathsf{B}xyz \Rightarrow x(yz)$ (*elementary compositor*), $\mathsf{C}_*xy \Rightarrow yx$ and $\mathsf{W}_*x \Rightarrow xx$. See Curry & al. (1958, p. 185) for these sets; William Craig (Curry & al. 1958, sect. 5H) proves important theorems on conditions for a set of combinators to be complete; in particular, at least two combinators are required.

[5]Craig (Curry & al. 1958, sect. 5H) discusses combinators in terms of various essential *effects* necessary to a computationally complete calculus: the *duplicative, cancellative, compositive* (parenthesis introducing) and *permutative effects*. In these terms we can see that K has the cancellative effect and S has the duplicative and compositive effects. The permutative effect is also provided by S, since $\mathsf{S}xyz \Rightarrow xz(yz)$ has put a copy of z in front of y.

[6]See MacLennan (1990, ch. 9) for a discussion of the Church-Rosser property and its proof.

the same answer. However, some orders may lead to nonterminating computations (which are the more interesting, from a biochemical standpoint). The Church-Rosser property is important for biochemical applications of combinatory logic, since it means that the rewrite rules can be applied in arbitrary orders, as they would be in a reaction vessel.

Our simulations use *normal order*, which basically means that a rules are applied as high up in the tree as possible; we apply them lower in the tree only when they cannot be applied higher up.[7] One can prove that normal order is maximally "safe" or "tolerant" in the sense that if any order terminates, then normal order will terminate. Conversely, if normal order leads to a nonterminating process when applied to some tree, then every order must lead to nontermination when applied to that tree. Therefore, our simulations are conservative, in the sense that whenever they result in unending growth, that unending growth is inevitable, since every order must produce it.

2 Investigation of Random SKI Trees

We begin with an investigation of the computational behavior of random SKI trees. The general approach is a follows: We generate a population of random SKI trees of limited size.[8] Then the SKI rewrite rules are applied to each tree. Most trees stop computing after a few steps (when none of the rewrite rules can be applied to them). Others, however, go on computing for some time, and may indeed be nonterminating. (We limit the processes to a fixed number of steps for each tree.)[9]

Since the behavior of the trees can be quite complicated, one way to approach them is by looking at the *size* of the trees, that is, the number of SKI atoms in them. (We measure size under the assumption of no sharing, so that S increases the size of the tree by duplicating its third argument.)

[7] See MacLennan (1990, pp. 322, 514) for a discussion of normal order (also called *applicative order* and *lazy evaluation*) and its properties.

[8] Typically, we generate trees randomly, with a certain probability p of a node being a leaf, which is equally likely to be S, K or I. Further, there is an absolute limit D on the depth of the tree. In the experiments described here, $p = 0.1$ and $D = 6$, so the maximum size of a tree is $2^6 = 64$.

[9] Of course, the undecidability of the halting problem implies that we cannot, in general, distinguish nonterminating computations from those that merely continue for very many steps.

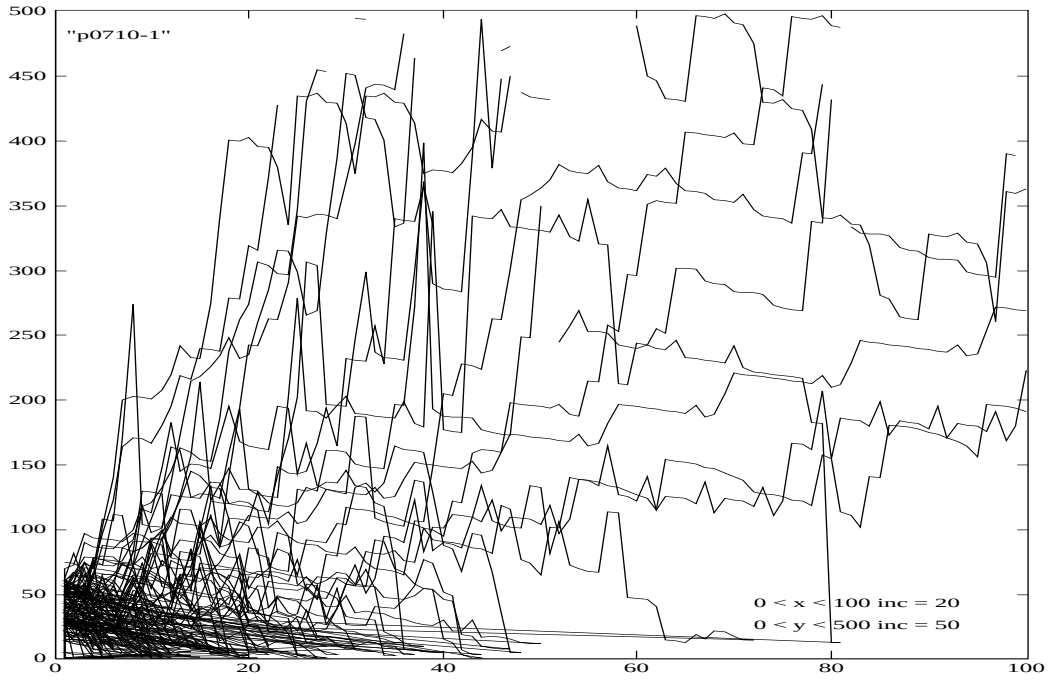


Figure 1: Size History of 100 Random SKI Trees (sizes ≤ 500).

For example, the following SKI tree appeared in one population of random trees:

```
(I I (S I (I S)) (S I (K S) (I S (I I)))
  (I I (S K) (K (S I)) (I S I S)))
```

The first 30 steps in its computation are shown in Appendix A. After 57 steps, there were 1216 atoms in the tree, after 63 steps there were 2308, and after 200 steps there were 1 506 854.

Figure 1 shows the *size history* of one population of 100 random SKI trees, limited in size to a depth less than 6, and thus to a size less than $2^6 = 64$ atoms; computation was limited to 100 steps. As can be seen, most of the trees are of monotonically decreasing size, and stop computing within 20 steps.[10] Some trees, however, show oscillation or more complex variations in their sizes, with overall trends that are either increasing or decreasing.

[10] The backtraces from the ends of terminating processes are artifactual from the plotting program. Fragments of graphs result from graphs that have gone out of the plot range $[0, 500]$.

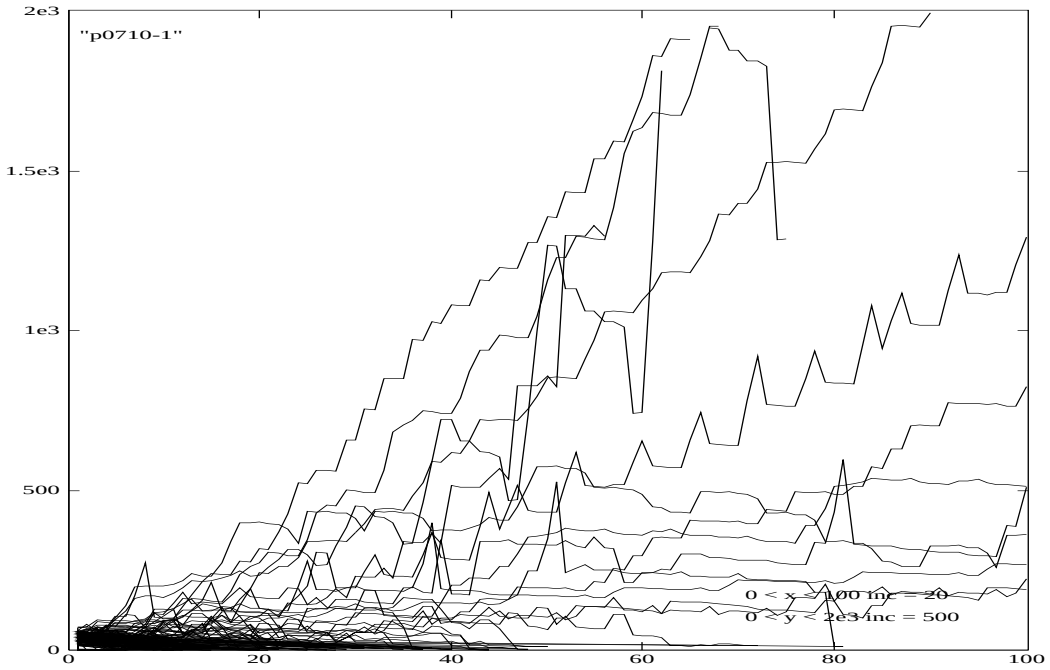


Figure 2: Size History of 100 Random SKI Trees (sizes ≤ 2000).

Within 40 steps some have grown to more than 500 atoms and so their histories have moved off the top of the graph.

In Fig. 2 we have "zoomed out" to show sizes up to 2000 atoms. We can now see that approximately 10% of the trees are growing at various rates. There is always some oscillation, but it is interesting that it is rarely purely periodic; if we look closely we always see some variation (often progressive) from cycle to cycle. This suggests that there are some complex processes taking place.

In Fig. 3 we have zoomed out again to show trees up to 10 000 in size. At this scale it appears that some of the trees are growing exponentially in size (as would be expected from the S combinator). Note the interesting sawtooth pattern superimposed on exponential growth as well as the exponentially increasing step pattern.

Finally, in Fig. 4 we zoom out to show trees up to size 60 000, and we can see a tree of exponentially increasing size, which had reached size 58 522 within the 100 steps to which it was restricted (it reached 94 642 on the next step).

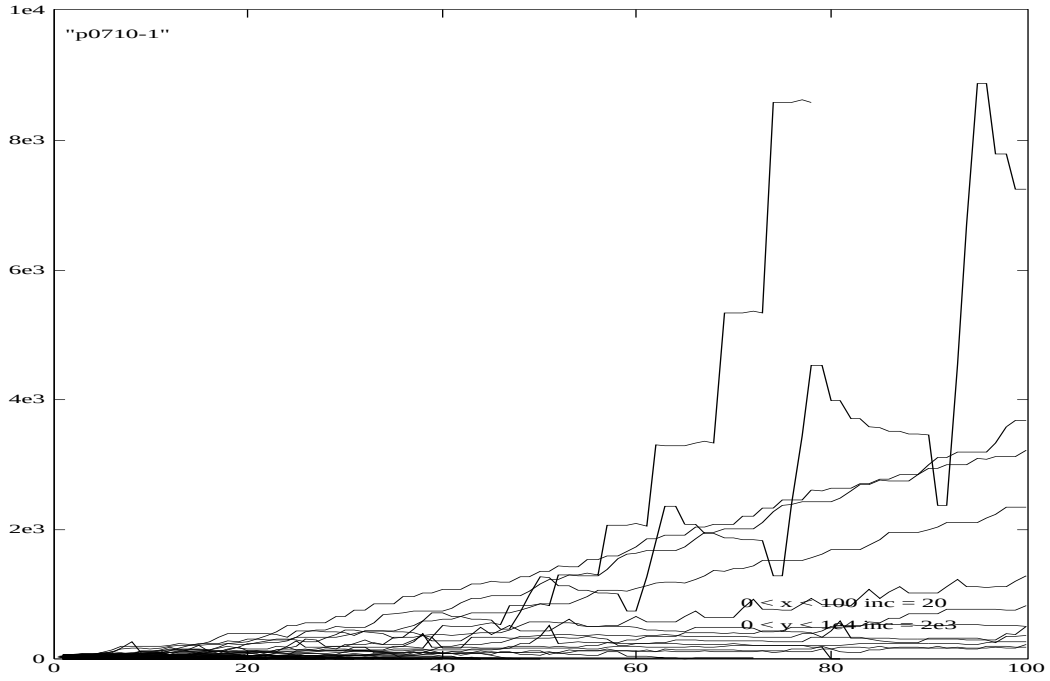


Figure 3: Size History of 100 Random SKI Trees (sizes $\leq$ 10 000).

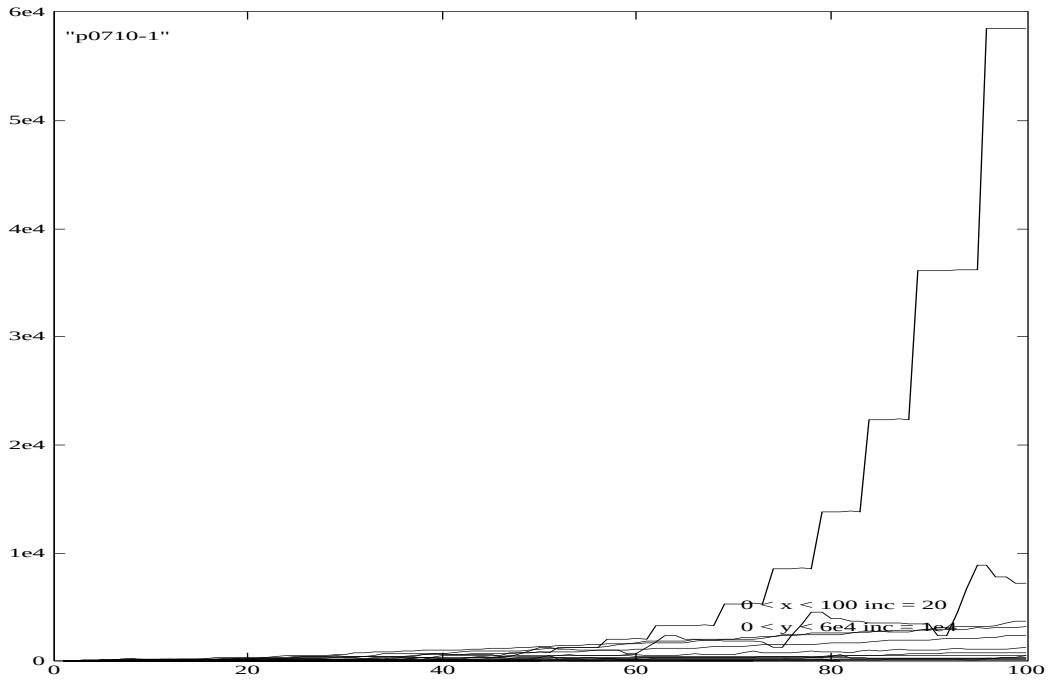


Figure 4: Size History of 100 Random SKI Trees (sizes $\leq$ 60 000).

We observe qualitatively similar behavior in other random populations. Therefore we can conclude that in a random population of random SKI trees, a reasonable fraction of them will exhibit complex processes leading to exponential growth in size.

3 S-augmentation and the λ Parameter

Since the S combinator is the only productive combinator in the SKI group, i.e., the only one that can cause the tree to increase in size, we expect the number of S combinators to be an important factor in achieving complex behavior (especially growth). S-augmentation is defined to be a process in which we increase the number of S combinators in an expression. In general, we are concerned with random S-augmentation, in which random (K or I) combinators are converted to S combinators. Similarly to Chris Langton's "edge of chaos" work (e.g. 1990), we define a parameter λ, which is the fraction of S combinators in a combinator tree. Thus, in S-augmentation experiments we run λ from 0 up to 1. In the experiments described in this report we restrict our attention to combinator *strings*, that is, to left-branching trees, which can be written linearly without parentheses.[11]

One such S-augmentation experiment is shown in Fig. 5. Here we have started with a single random KI-string of length 50. It has been subjected to four different random S-augmentation processes. That is, in each process, K and I combinators are randomly converted to S combinators, but different K and I combinators are converted in each of the four histories. We see that the process length increases with λ up to a fairly sharp phase transition at $\lambda \approx 0.5$. Above the phase transition, the processes are (apparently) non-terminating (that is, they hit the imposed limit of 150 steps). Nevertheless, further S-augmentation can sometimes lead to terminating processes, as we see in two of the dotted histories (`slam50-10c` and `slam50-10d`). This is because the dynamic behavior of the SKI string can depend sensitively on the specific order of the combinators; nevertheless, from a statistical standpoint, it depends on λ, the fraction of S combinators.

Figure 6 displays the results of another S-augmentation experiment, in this case starting with a random KI-string of length 100. Qualitatively similar

[11] Although strings (as opposed to trees) are interesting from a biochemical standpoint, arbitrary trees should also be investigated. This preliminary investigation limited itself to strings as a matter of programming convenience.

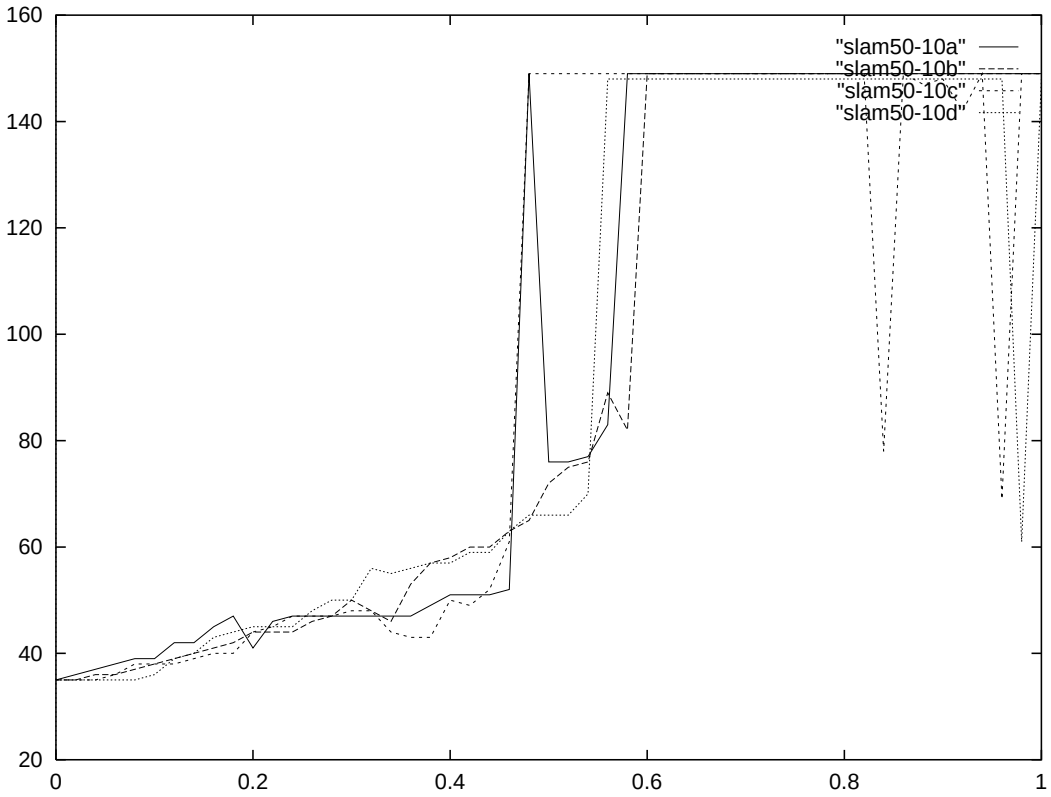


Figure 5: Four random S-augmentation histories starting from the same random KI-string of length 50. The graph shows the number of steps to termination (limited to 150) versus λ, the fraction of S combinators in the string.

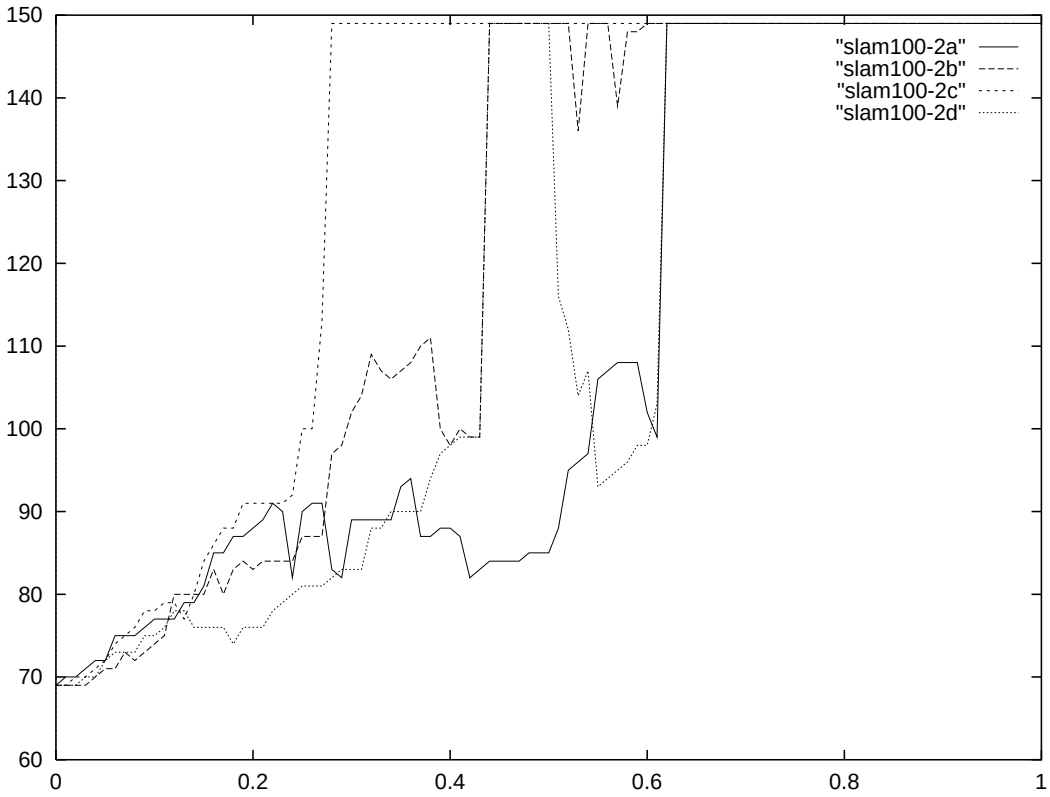


Figure 6: Four random S-augmentation histories starting from the same random KI-string of length 100. The graph shows the number of steps to termination (limited to 150) versus λ, the fraction of S combinators in the string.

behavior can be observed, although the phase transition is not so well defined, varying in the range $0.2 \leq \lambda \leq 0.6$. What this suggests is that some strings can be more sensitive than others to the placement of the S combinators, as opposed to just their numbers.

Of course it is not obvious that the λ parameter as defined (fraction of S combinators) is the most relevant; it was suggested by Langton's λ. For example, since the S and K combinators are in competition in a sense (the K throwing information away and the S duplicating it), we might suppose that the ratio of S to K combinators is a key variable. Alternate parameterizations of the trees deserve systematic investigation.

As a step in this direction, we have investigated the interaction of S with each of the other combinators in S-augmentation starting from homogeneous strings of just Ks or just Is. It is easy to prove that a string of K combinators of length n will reduce to either one or two Ks in approximately $n/2$ steps.[12] It is also easy to prove that a string of I combinators of length n will reduce to a single I combinator in $n-1$ steps. Finally, one can prove (not so easily) that a string of S combinators of length n will reduce to an expression of the form

$$\mathsf{SS}(\mathsf{SS}(\mathsf{SS}(\cdots(\mathsf{SS}(\mathsf{S}^m))\cdots))),$$

where $m = 2+(n \bmod 2)$, in $(n-1)^2/4$ steps if n is even, and in $(n-1)(n-3)/4$ steps if n is odd; that is, in approximately $n^2/4$ steps. In particular, if $n = 25$, computation stops in 132 steps.

Figure 7 shows four random S-augmentations starting from a string of 25 Ks. As expected, when $\lambda = 0$ the string terminates in 12 steps. The number of steps to termination increases very gradually until the string is almost all Ss, at which point termination time jumps to 132 steps. (Indeed, if there was more than one K in the string, $\lambda < 0.96$, it terminated in less than 132 steps.) So, in a "competition" between K and S combinators, we see that the K is "stronger." There is no evidence of unlimited growth in SK strings.[13]

Figure 8 shows four random S-augmentations starting from a string of 25 Is. As predicted, the steps-to-termination begins at 24, but we see a sudden phase transition to apparently unlimited growth in the region $0.25 < \lambda < 0.5$.

[12] Specifically, $\mathsf{K}^{2m+1} \Rightarrow \mathsf{K}$ in m steps, and $\mathsf{K}^{2m+2} \Rightarrow \mathsf{KK}$ in m steps, for $m \geq 0$.

[13] On the other hand, as mentioned in footnote 3, the S and K combinators are sufficient for universal computation, since I can be defined in terms of S and K. Out of the eight possible SK-strings of length three, two are equivalent to I (i.e. SKK and SKS), but they may still be relatively unlikely, since it takes three combinators to get the effect of an I, but only one to get the effect of an S or K. More analysis will resolve this paradox.

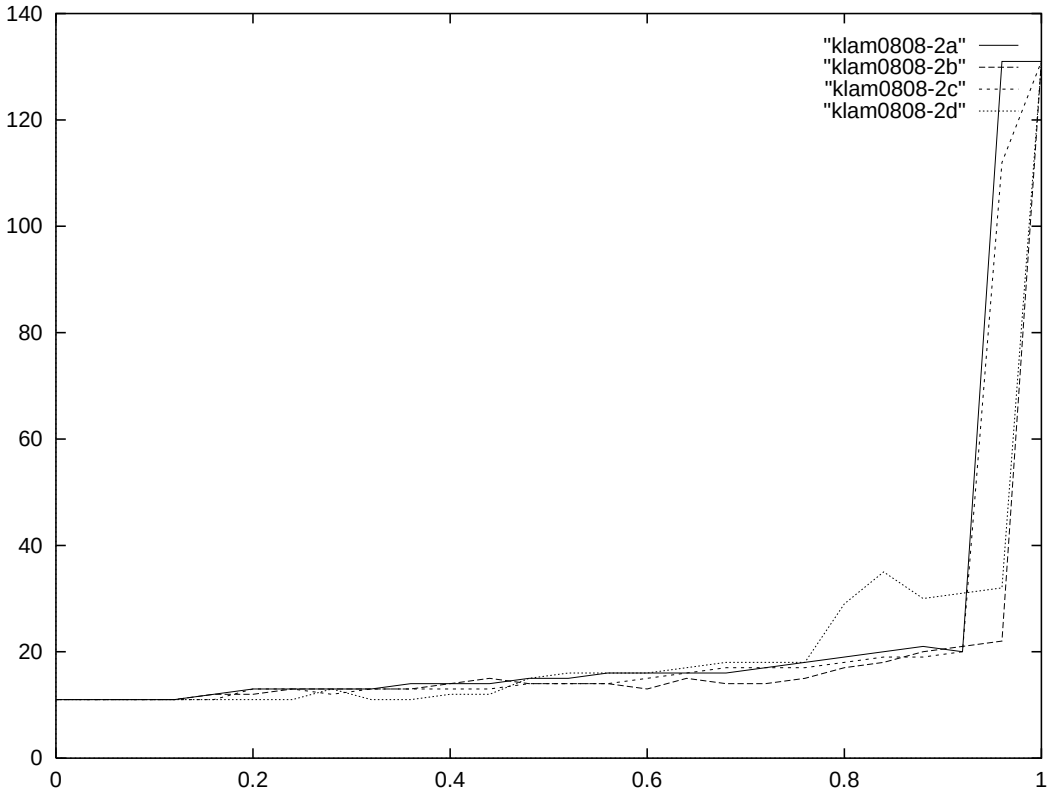


Figure 7: Four random S-augmentation histories starting from a string of 25 K combinators. Runs were limited to 200 steps.

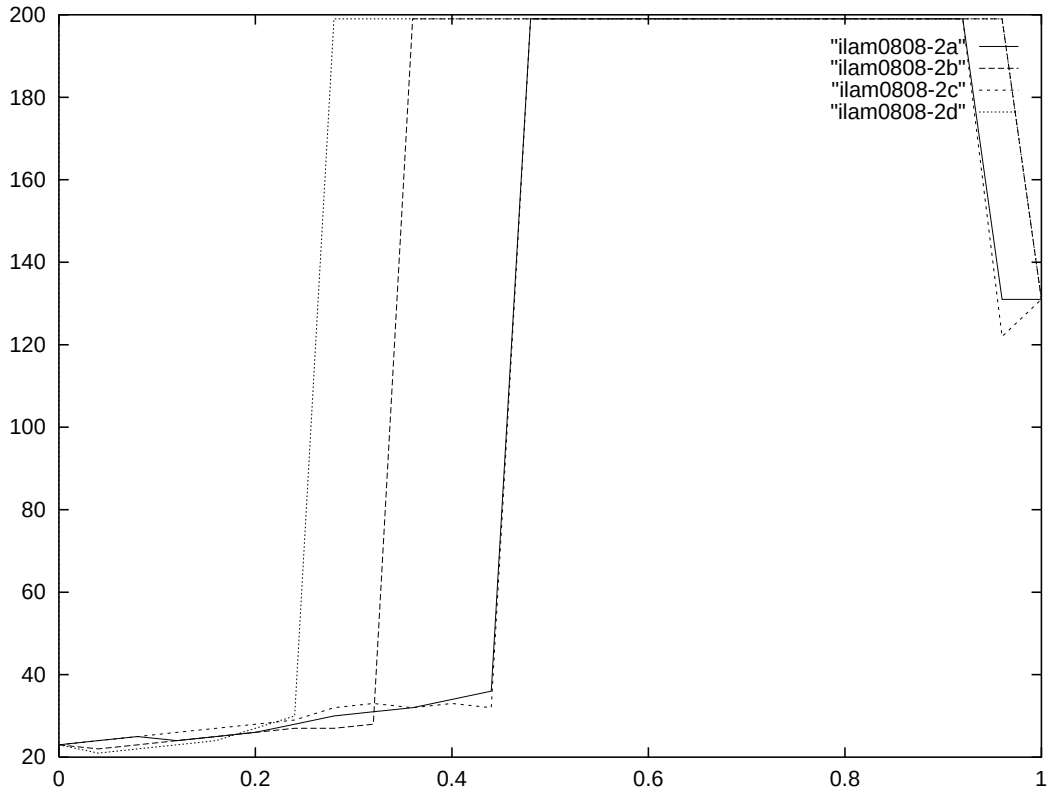


Figure 8: Four random S-augmentation histories starting from a string of 25 I combinators. Runs were limited to 200 steps.

However, at high λ values (greater than 0.92) the steps to termination must drop back to the 132 required for 25 S's.

By combining the insights from Figs. 7 and 8 we can begin to understand some of the behavior we saw in S-augmentations starting from random KI strings (e.g. Fig. 5). For $\lambda = 0$, termination takes about $\frac{3}{4}n$ steps on the average (i.e., 37 steps for $n = 50$ as in Fig. 5). For strings dominated by K and I, the computation length increases gradually with λ. At some critical values of λ the interaction of S and I combinators results in long computations. At very high levels of λ there may not be sufficient Is to sustain long computation, and computation will be dominated by the S, leading to termination in a moderate number of steps ($\approx n^2/4$). However, without more systematic experiments, this explanation is largely speculation.

4 Conclusions

Several tentative conclusions can be drawn from these exploratory studies. First, the simple combinatorial processes represented by the SKI combinators can lead to exponential growth and complex behavior in a significant fraction of populations of random SKI-combinator trees. Second, from a statistical perspective, the behavior of random SKI strings depends on the relative proportion of the combinators. Specifically, with an increasing proportion of S combinators, there seems to be a fairly abrupt phase transition from terminating processes to complex growth processes. This phase transition seems to be related to the "edge of chaos" explored by Chris Langton, and is characterized by a related parameter (λ).

Of course, these experiments leave many questions unanswered, and suggest paths for more systematic esperimental investigations. For example:

1. What parameter(s) best characterize the statistical behavior of these random populations? The "competition" between S and K (one duplicating, the other deleting) suggests that the ratio of their concentrations should be significant, but our experiments indicate that the ratio of S to I concentration may be more important in characterizing the transition to exponential growth. It would be worthwhile to do three-dimensional plots exploring all relative concentrations of S, K and I in the "soup."

2. Our S-augmentation experiments have been limited to combinator *strings*;

it is necessary to see if random combinator *trees* behave similarly.

3. We also need to explore a greater variety of string lengths (or tree sizes) to see whether or not phase transition points are independent of these parameters.

4. Observation of individual growth processes (such as that in App. A), as well as the pattern of size increases (e.g. Figs. 1 and 2), suggests that the trees become self-embedding in complex ways. Analogously to the way that Langton measured cycle-length in his cellular automata, we would like have some guage of self-embedding depth. This suggests measures such as fractal dimension and tree-oriented autocorrelations, but the best measure is far from obvious.

5. Although the SKI combinators are computationally complete, it is not obvious that that is necessary for exponential growth or relevant in a biochemical context. For example the W combinator, like S, can duplicate information, so we might consider the WKI triad, which is not computationally complete. In fact it doesn't seem very interesting, since the statistically common string WWW leads to a nonterminating but nongrowing process (WWW $\Rightarrow$ WWW $\Rightarrow \cdots$). However, other sets of combinators may lead to more interesting behavior.

6. It is possible to add "inert" atoms (i.e. atoms for which there are no rewrite rules, thus corresponding to purely passive data) to the "soup." It is expected that the concentrations of only the active ingredients will be significant to the behavior, but this needs to be checked.

7. Certainly, one of the biggest issues is the relevance of all this to biochemistry, which depends on their being biochemical processes that can be reasonably modeled as the SKI or other combinators. (On the other hand, even if these studies have no applications to biological self-organization, they are still interesting from the perspective of complex systems and emergent computation.)

5 Acknowledgements

The Institute for Advanced Studies of the Collegium Budapest provided an ideal environment for this exploratory work. I am grateful to Jim Hurford and

Simon Kirby, and especially to Eörs Szathmáry and Guenter von Kiedrowsky, for helpful discussions.

6 Bibliography

1. Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (revised edition). North-Holland.

2. Curry, Haskell B., Feys, R. & Craig, W. (1958). *Combinatory Logic, Vol. I.* North-Holland.

3. Fasel, Joseph H., & Keller, Robert M. (Eds.) (1987). *Graph Reduction.* Lecture Note in Computer Science 279. Springer-Verlag.

4. Fontana, W., & Buss, L. W. (1994). 'The arrival of the fittest': toward a theory of biological organization. *Bull. Math. Biol.* **56**: 1–64.

5. Langton, C. G. (1990). Computation at the edge of chaos: phase transitions and emergent computation. *Physica D* **49**: 12–37; reprinted in Stephanie Forrest (Ed.), *Emergent Computation*, North-Holland, 12–37.

6. MacLennan, Bruce J. (1990). *Functional Programming: Practice and Theory.* Addison-Wesley.

7. Szathmáry, Eörs (1995). A classification of replicators and lambda-calculus models of biological organization. *Proc. Royal. Soc. London B* **260**: 279–286.

8. Szathmáry, Eörs (1997). The first two billion years. *Nature* **387**: 662–663.

9. Turner, D. A. (1979). A new implementation technique for applicative languages. *Software — Practice and Experience* **9**: 31–49.

A Example Random SKI Tree Exhibiting Exponential Growth

In this appendix we show the first 30 steps in the computation of the tree

(I I (S I (I S)) (S I (K S) (I S (I I)))
  (I I (S K) (K (S I)) (I S I S)))

which appeared in a population of 100 random trees.

```
Step 1 =
      (I I (S I (I S)) (S I (K S) (I S (I I)))
         (I I (S K) (K (S I)) (I S I S)))
Step 2 =
      (I (S I (I S)) (S I (K S) (I S (I I)))
         (I I (S K) (K (S I)) (I S I S)))
Step 3 =
      (S I (I S) (S I (K S) (I S (I I)))
         (I I (S K) (K (S I)) (I S I S)))
Step 4 =
      (I (S I (K S) (I S (I I))) (I S (S I (K S) (I S (I I))))
         (I I (S K) (K (S I)) (I S I S)))
Step 5 =
      (S I (K S) (I S (I I)) (I S (S I (K S) (I S (I I))))
         (I I (S K) (K (S I)) (I S I S)))
Step 6 =
      (I (I S (I I)) (K S (I S (I I)))
         (I S (I (I S (I I)) (K S (I S (I I)))))
         (I I (S K) (K (S I)) (I S I S)))
Step 7 =
      (I S (I I) (K S (I S (I I))) (I S (I S (I I) (K S (I S (I I)))))
         (I I (S K) (K (S I)) (I S I S)))
Step 8 =
      (S (I I) (K S (S (I I))) (I S (S (I I) (K S (S (I I)))))
         (I I (S K) (K (S I)) (I S I S)))
Step 9 =
      (I I (I S (S (I I) (K S (S (I I)))))
         (K S (S (I I)) (I S (S (I I) (K S (S (I I))))))
         (I I (S K) (K (S I)) (I S I S)))
```

```
Step 10 =
      (I (I S (S (I I) (K S (S (I I)))))
         (K S (S (I I)) (I S (S (I I) (K S (S (I I))))))
         (I I (S K) (K (S I)) (I S I S)))
Step 11 =
      (I S (S (I I) (K S (S (I I))))
         (K S (S (I I)) (I S (S (I I) (K S (S (I I))))))
         (I I (S K) (K (S I)) (I S I S)))
Step 12 =
      (S (S (I I) (K S (S (I I))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I))))))
         (I I (S K) (K (S I)) (I S I S)))
Step 13 =
      (S (I I) (K S (S (I I))) (I I (S K) (K (S I)) (I S I S))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (I I (S K) (K (S I)) (I S I S))))
Step 14 =
      (I I (I I (S K) (K (S I)) (I S I S))
         (K S (S (I I)) (I I (S K) (K (S I)) (I S I S)))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (I I (S K) (K (S I)) (I S I S))))
Step 15 =
      (I (I I (S K) (K (S I)) (I S I S))
         (K S (S (I I)) (I I (S K) (K (S I)) (I S I S)))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (I I (S K) (K (S I)) (I S I S))))
Step 16 =
      (I I (S K) (K (S I)) (I S I S)
         (K S (S (I I)) (I I (S K) (K (S I)) (I S I S)))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (I I (S K) (K (S I)) (I S I S))))
Step 17 =
      (I (S K) (K (S I)) (I S I S)
         (K S (S (I I)) (I (S K) (K (S I)) (I S I S)))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (I (S K) (K (S I)) (I S I S))))
Step 18 =
      (S K (K (S I)) (I S I S)
```

```
         (K S (S (I I)) (S K (K (S I)) (I S I S)))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (S K (K (S I)) (I S I S))))
Step 19 =
      (K (I S I S) (K (S I) (I S I S))
         (K S (S (I I)) (K (I S I S) (K (S I) (I S I S))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (I S I S) (K (S I) (I S I S)))))
Step 20 =
      (I S I S (K S (S (I I)) (K (I S I S) (K (S I) (I S I S))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (I S I S) (K (S I) (I S I S)))))
Step 21 =
      (S I S (K S (S (I I)) (K (S I S) (K (S I) (S I S))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S)))))
Step 22 =
      (I (K S (S (I I)) (K (S I S) (K (S I) (S I S))))
         (S (K S (S (I I)) (K (S I S) (K (S I) (S I S)))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S)))))
Step 23 =
      (K S (S (I I)) (K (S I S) (K (S I) (S I S)))
         (S (K S (S (I I)) (K (S I S) (K (S I) (S I S)))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S)))))
Step 24 =
      (S (K (S I S) (K (S I) (S I S)))
         (S (S (K (S I S) (K (S I) (S I S)))))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S)))))
Step 25 =
      (K (S I S) (K (S I) (S I S))
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S))))
         (S (S (K (S I S) (K (S I) (S I S))))
            (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S))))))
```

```
Step 26 =
      (S I S
         (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S))))
         (S (S (K (S I S) (K (S I) (S I S)))
            (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S)))))))
Step 27 =
      (I (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
            (K (S I S) (K (S I) (S I S))))
         (S (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S)))))
         (S (S (K (S I S) (K (S I) (S I S))))
            (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S))))))
Step 28 =
      (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
         (K (S I S) (K (S I) (S I S)))
         (S (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S)))))
         (S (S (K (S I S) (K (S I) (S I S))))
            (K S (S (I I)) (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S))))))
Step 29 =
      (S (S (S (I I) (K S (S (I I))))) (K (S I S) (K (S I) (S I S)))
         (S (S (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S)))))
         (S (S (K (S I S) (K (S I) (S I S))))
            (S (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S))))))
Step 30 =
      (S (S (I I) (K S (S (I I))))
         (S (S (S (S (I I) (K S (S (I I)))))
               (K (S I S) (K (S I) (S I S)))))
         (K (S I S) (K (S I) (S I S))
            (S (S (S (S (I I) (K S (S (I I)))))
                  (K (S I S) (K (S I) (S I S))))))
         (S (S (K (S I S) (K (S I) (S I S))))
```

```
(S (S (S (I I) (K S (S (I I)))))
   (K (S I S) (K (S I) (S I S))))))
```

B LISP Definitions for SKI Reduction

This is the basic LISP program for doing SKI computations, generating SKI trees, etc.

```
;       SKI Size II
;       Routines for studying the reduction of
;       random SKI combinator trees by keeping track of their size.
;       This version does not share subtrees.
;       Bruce MacLennan
;       Collegium Budapest
;       July 1997

;       (limit-run tree limit)
;       Complete normal-order reduction of an untagged combinator tree,
;       which is not allowed to continue past limit passes.
;       A global variable, no-reduction, is set to t if the
;       reduction completed normally, and to nil if it hit the limit.
;       Prints beginning and final trees in
;       parenthesis-free form.
;       Optionally (plot-size) writes plot file containing tree
;       size (atom count) after each step.
;       The annotated tree is left in a variable called "root."
;       Note that the tree is modified by the reduction.
;       A copy of the original, unmodified annotated tree is left
;       in orig-tree.

(defun limit-run (tree limit)
  (setq orig-tree (copy-tree tree))
  (setq root (cons (size-annotate tree) nil))
  (print (list 'beginning (flatten (car root))))
  (if pr (print (list 'beginning 'tree (car root))))
  (continue-running 1 limit))

(defun continue-running (n limit)
  (setq no-reduction t)
  (rplaca root (reduce-tree (car root)))
  (setq passes n)
  (if pr (print (list 'pass n '= (flatten (car root))))))
```

```
  (cond
   (plot-size
    (print n plot-file)
    (princ "     " plot-file)
    (princ (size (car root)) plot-file)
    ))                                                          40
  (cond ((and (not no-reduction) (<= n limit))
         (continue-running (1+ n) limit))
        (t
         (if pr (print (list
                        (if no-reduction 'completed 'terminated)
                        (flatten (car root)))))
         root))
  )

;       (reduce-tree atree)                                     50
;       Attempts to perform one normal-order reduction on the annotated tree
;       and returns the (possibly modified) annotated tree.
;       If a reduction is performed, then the global
;       variable no-reduction is set to nil.
;       (It is assumed to have been previously set to t.)

(defun reduce-tree (tree)
  (if dpr (print (list 'reducing tree)))
  (let ((rtree (try-reduction tree)))
       (cond                                                    60
        ((and no-reduction (listp (untag rtree)) (untag rtree))
         (if dpr (print (list 'going 'left (tagcar rtree))))
         (rplaca (untag rtree) (reduce-tree (tagcar rtree)))
         (rplaca rtree
                 (+ (size (tagcar rtree)) (size (tagcdr rtree))))
         (cond
          ((and no-reduction (tagcdr rtree))
           (if dpr (print (list 'going 'right (tagcdr rtree))))
           (rplacd (untag rtree) (reduce-tree (tagcdr rtree)))
           (rplaca rtree                                        70
                   (+ (size (tagcar rtree)) (size (tagcdr rtree)))))
          ))
```

```
        )
       (if (and dpr (not no-reduction))
           (print (list 'reduced rtree)))
       rtree)
  )

;       (try-reduction atree)
;       Attempts to apply a single reduction to the root                    80
;       of the annotated tree.
;       If successful, no-reduction is set to nil.
;       In either case the (possibly modified) annotated tree is returned.

(defun try-reduction (L)
  (cond
   ((atom (untag L)) L)
   ((null L) L)
   ;     Process I combinator:  Ix => x
   ((and (eq (untag (tagcar L)) 'I) (tagcdr L))                             90
    (setq no-reduction nil)
    (if dpr (print (list 'I 'found L '=> (tagcdr L))))
    (tagcdr L)
    )
   ;     Process K combinator:  Kxy => x
   ((and
     (listp (untag (tagcar L)))
     (eq (untag (tagcar (tagcar L))) 'K))
    (setq no-reduction nil)
    (if dpr (print (list 'K 'found L '=> (tagcdr (tagcar L)))))             100
    (tagcdr (tagcar L))
    )
   ;     Process S combinator:  Sxyz => xz(yz)
   ((and
     (listp (untag (tagcar L)))
     (listp (untag (tagcar (tagcar L))))
     (eq (untag (tagcar (tagcar (tagcar L)))) 'S))
    (setq no-reduction nil)
    (if dpr (print (list 'S 'found L)))
    (let* ((M (untag (tagcar L)))                                           110
```

```
            (N (untag (car M)))
            (z (tagcdr L))
            (a (+ (size (cdr M)) (size z)))
            (b (+ (size (cdr N)) (size z)))
            )
           (rplaca (untag L) (cons b (cons (cdr N) z)))
           (rplacd (untag L) (cons a (cons (cdr M) (copy-tree z))))
           (rplaca L (+ a b))
           )
     (if dpr (print (list '=> L)))                                    120
     L)
    (t L)
    ))

;       (flatten atree)
;       Converts an annotated combinator tree into a list that
;       has redundant (left associating) parentheses eliminated.
;       This is a convenient way to print combinator trees.
;       (flatten (bintree S)) is the same as S, except that
;       redundant parentheses will have been eliminated.           130

(defun flatten (tree)
  (if (atom (untag tree))
      (untag tree)
      (reverse (flat tree))))

(defun flat (tree)
  (if (atom (untag tree))
      (list (untag tree))
      (cons (dflat (tagcdr tree)) (flat (tagcar tree)))))          140

(defun dflat (tree)
  (if (atom (untag tree))
      (untag tree)
      (reverse (flat tree))))

;       (bintree string)
;       Converts a string representing a combinator
```

```
;       formula, such as (S K K (K (S K) K) S),
;       into the corresponding binary tree.                          150
;       Note that (bintree (flatten T)) = T.

(defun bintree (string)
  (if (atom string)
      string
      (bintree-list string)))

(defun bintree-list (string)
  (if (null (cdr string))
      (bintree (car string))                                         160
      (bintree-op string)))

(defun bintree-op (string)
  (cons (bintree (butlast string))
        (bintree (car (last string)))))

;       Auxiliary function for generating random trees or strings.

(defun combnum (n)
  (cond ((eq n 0) 'I)                                                170
        ((eq n 1) 'K)
        ((eq n 2) 'S)
        ))

;       (ranstring n)
;       Generate a random, parenthesis-free combinator
;       string (in tree form) of length n.
;       This is a left branching tree, which can be written
;       without parentheses.
                                                                     180
(defun ranstring (n)
  (if (eq n 2)
      (cons (combnum (random 3)) (combnum (random 3)))
      (cons (ranstring (1- n)) (combnum (random 3)))))

;       (rantree p md)
```

```
;       Generate a random combinator tree with
;       application probability p (values such as 0.9 work well)
;       and maximum depth md.  The application probability is
;       the probability a node will be interior, rather than a leaf.
;       Such a tree will have at most 2^md nodes.

(defun rantree (p md)
  (if (or (> (* (random 1000) 0.001) p) (zerop md))
      (combnum (random 3))
      (cons (rantree p (1- md)) (rantree p (1- md)))))

;       (initran n)
;       Initializes random number generator by calling
;       it n times.  When executed in a fresh LISP environment,
;       this allows repeatable results.

(defun initran (n) (dotimes (k (1+ n)) (random 1)) 'done)

;       Print control variables:
;       Setting pr causes printing of the tree after each reduction.
;       Setting dpr causes additional, extensive debugging information
;       to be printed.
;       Setting plot-size causes tree size to be written to plot-file
;       after each step.

(setq pr nil)
(setq dpr nil)
(setq plot-size nil)

;       (limit-multirun seed n md pf limit)
;       Multiple run facility, with a limit on number of steps.
;       Generates n trees randomly (seeded by sd), given maximum depth md.
;       (The nonleaf probability is fixed at 0.9.)
;       While running displays (R A P) on console, where R is run number,
;       A is tree atom count and and P is number of passes.
;       Writes atom-count / passes summary to a file called
;       pf.
;       Example:  (limit-multirun 37 100 5 "plot" 100)
```

```
(defun limit-multirun (seed n md pf limit)
  (setq plot-file (open pf :direction :output))
  (initran seed)
  (dotimes (k n)
           (let* ((rt (rantree 0.9 md))                              230
                  (len (atoms (flatten rt))))
                 (limit-run rt limit)
                 (print (list k len (1- passes)))
                 (print len plot-file)
                 (princ "     " plot-file)
                 (princ (1- passes) plot-file)
                 ))
  (close plot-file)
  'done
  )                                                                  240

;       (size-multirun seed n md pf limit)
;       Multiple run facility with plotting of running
;       tree size (atom count) and limit on number of steps.
;       It keeps track of the sizes during reduction by using
;       size-annotated trees.
;       Generates n trees randomly (seeded by sd), given maximum
;       depth md.  (The nonleaf probability is fixed at 0.9.)
;       After each run, prints tree number, number of passes
;       and final tree size to console.                            250
;       Writes step number and tree size to plot file;
;       data from each tree is separated by a blank line.

(defun size-multirun (seed n md pf limit)
  (setq plot-file (open pf :direction :output))
  (setq plot-size t)
  (princ "#params " plot-file)
  (prin1 (list seed n md pf limit) plot-file)
  (initran seed)
  (dotimes (k n)                                                     260
           (let ((rt (rantree 0.9 md)))
                (limit-run rt limit)
```

```
                (print (list 'tree k
                             'had (1- passes) 'passes
                             'size (size (car root))))
                )
            (terpri plot-file)
            )
  (close plot-file)
  'done                                                          270
  )

;       (atoms tree)
;       Counts the number of atoms in a tree.

(defun atoms (tree)
  (cond
   ((null tree) 0)
   ((atom tree) 1)
   (t (+ (atoms (car tree)) (atoms (cdr tree))))                 280
   ))

;       (size-annotate tree)
;       Adds size annotations to each node of a binary tree.

(defun size-annotate (tree)
  (if (atom tree)
      (cons 1 tree)
      (let ((left (size-annotate (car tree)))
            (right (size-annotate (cdr tree)))                  290
            )
           (cons (+ (size left) (size right))
                 (cons left right))
           )))

;       (untag atree)
;       Removes size-tag from an annotated tree.

(defun untag (L) (cdr L))
                                                               300
```

```
;       (tagcar atree)
;       (tagcdr atree)
;       Performs car or cdr on an annotated binary tree.

(defun tagcar (L) (cadr L))
(defun tagcdr (L) (cddr L))

;       (size atree)
;       Return size-tag of an annotated tree.
                                                                    310
(defun size (L) (car L))
```

C LISP Definitions for SKI Reduction with Size Computation

Since the SKI trees can grow exponentially in size, it becomes too expensive to compute their sizes by simply scanning the tree and counting at each step. Therefore, a modification of the program was developed that keeps track of the tree's size during computation. Each node is now represented by a LISP "dotted pair" $(S \ . \ N)$, where N is the bare node [i.e. an SKI leaf or a dotted pair $(X \ . \ Y)$ representing the function application (XY)], and S is the size of this node. Of course, the size computations produced by this program where checked against those produced by the original program.

```
;       SKI Size II
;       Routines for studying the reduction of
;       random SKI combinator trees by keeping track of their size.
;       This version does not share subtrees.
;       Bruce MacLennan
;       Collegium Budapest
;       July 1997

;       (limit-run tree limit)
;       Complete normal-order reduction of an untagged combinator tree,
;       which is not allowed to continue past limit passes.
;       A global variable, no-reduction, is set to t if the
;       reduction completed normally, and to nil if it hit the limit.
;       Prints beginning and final trees in
;       parenthesis-free form.
;       Optionally (plot-size) writes plot file containing tree
;       size (atom count) after each step.
;       The annotated tree is left in a variable called "root."
;       Note that the tree is modified by the reduction.
;       A copy of the original, unmodified annotated tree is left      20
;       in orig-tree.

(defun limit-run (tree limit)
  (setq orig-tree (copy-tree tree))
  (setq root (cons (size-annotate tree) nil))
  (print (list 'beginning (flatten (car root))))
```

```
  (if pr (print (list 'beginning 'tree (car root))))
  (continue-running 1 limit))

(defun continue-running (n limit)                                    30
  (setq no-reduction t)
  (rplaca root (reduce-tree (car root)))
  (setq passes n)
  (if pr (print (list 'pass n '= (flatten (car root)))))
  (cond
   (plot-size
    (print n plot-file)
    (princ "     " plot-file)
    (princ (size (car root)) plot-file)
    ))                                                               40
  (cond ((and (not no-reduction) (<= n limit))
         (continue-running (1+ n) limit))
        (t
         (if pr (print (list
                        (if no-reduction 'completed 'terminated)
                        (flatten (car root)))))
         root))
  )

;       (reduce-tree atree)                                            50
;       Attempts to perform one normal-order reduction on the annotated tree
;       and returns the (possibly modified) annotated tree.
;       If a reduction is performed, then the global
;       variable no-reduction is set to nil.
;       (It is assumed to have been previously set to t.)

(defun reduce-tree (tree)
  (if dpr (print (list 'reducing tree)))
  (let ((rtree (try-reduction tree)))
       (cond                                                         60
        ((and no-reduction (listp (untag rtree)) (untag rtree))
         (if dpr (print (list 'going 'left (tagcar rtree))))
         (rplaca (untag rtree) (reduce-tree (tagcar rtree)))
         (rplaca rtree
```

```
                    (+ (size (tagcar rtree)) (size (tagcdr rtree))))
          (cond
           ((and no-reduction (tagcdr rtree))
            (if dpr (print (list 'going 'right (tagcdr rtree))))
            (rplacd (untag rtree) (reduce-tree (tagcdr rtree)))
            (rplaca rtree                                              70
                    (+ (size (tagcar rtree)) (size (tagcdr rtree)))))
           ))
          )
         (if (and dpr (not no-reduction))
             (print (list 'reduced rtree)))
         rtree)
  )

;        (try-reduction atree)
;        Attempts to apply a single reduction to the root              80
;        of the annotated tree.
;        If successful, no-reduction is set to nil.
;        In either case the (possibly modified) annotated tree is returned.

(defun try-reduction (L)
  (cond
   ((atom (untag L)) L)
   ((null L) L)
   ;     Process I combinator:  Ix => x
   ((and (eq (untag (tagcar L)) 'I) (tagcdr L))                        90
    (setq no-reduction nil)
    (if dpr (print (list 'I 'found L '=> (tagcdr L))))
    (tagcdr L)
    )
   ;     Process K combinator:  Kxy => x
   ((and
     (listp (untag (tagcar L)))
     (eq (untag (tagcar (tagcar L))) 'K))
    (setq no-reduction nil)
    (if dpr (print (list 'K 'found L '=> (tagcdr (tagcar L)))))      100
    (tagcdr (tagcar L))
    )
```

```
    ;     Process S combinator:  Sxyz => xz(yz)
    ((and
      (listp (untag (tagcar L)))
      (listp (untag (tagcar (tagcar L))))
      (eq (untag (tagcar (tagcar (tagcar L)))) 'S))
     (setq no-reduction nil)
     (if dpr (print (list 'S 'found L)))
     (let* ((M (untag (tagcar L)))                                    110
            (N (untag (car M)))
            (z (tagcdr L))
            (a (+ (size (cdr M)) (size z)))
            (b (+ (size (cdr N)) (size z)))
            )
           (rplaca (untag L) (cons b (cons (cdr N) z)))
           (rplacd (untag L) (cons a (cons (cdr M) (copy-tree z))))
           (rplaca L (+ a b))
           )
     (if dpr (print (list '=> L)))                                    120
     L)
    (t L)
    ))

;       (flatten atree)
;       Converts an annotated combinator tree into a list that
;       has redundant (left associating) parentheses eliminated.
;       This is a convenient way to print combinator trees.
;       (flatten (bintree S)) is the same as S, except that
;       redundant parentheses will have been eliminated.           130

(defun flatten (tree)
  (if (atom (untag tree))
      (untag tree)
      (reverse (flat tree))))

(defun flat (tree)
  (if (atom (untag tree))
      (list (untag tree))
      (cons (dflat (tagcdr tree)) (flat (tagcar tree)))))          140
```

```
(defun dflat (tree)
  (if (atom (untag tree))
      (untag tree)
      (reverse (flat tree))))

;         (bintree string)
;         Converts a string representing a combinator
;         formula, such as (S K K (K (S K) K) S),
;         into the corresponding binary tree.                    150
;         Note that (bintree (flatten T)) = T.

(defun bintree (string)
  (if (atom string)
      string
      (bintree-list string)))

(defun bintree-list (string)
  (if (null (cdr string))
      (bintree (car string))                                     160
      (bintree-op string)))

(defun bintree-op (string)
  (cons (bintree (butlast string))
        (bintree (car (last string)))))

;         Auxiliary function for generating random trees or strings.

(defun combnum (n)
  (cond ((eq n 0) 'I)                                            170
        ((eq n 1) 'K)
        ((eq n 2) 'S)
        ))

;         (ranstring n)
;         Generate a random, parenthesis-free combinator
;         string (in tree form) of length n.
;         This is a left branching tree, which can be written
```

```
;       without parentheses.
                                                                    180
(defun ranstring (n)
  (if (eq n 2)
      (cons (combnum (random 3)) (combnum (random 3)))
      (cons (ranstring (1- n)) (combnum (random 3)))))

;       (rantree p md)
;       Generate a random combinator tree with
;       application probability p (values such as 0.9 work well)
;       and maximum depth md.  The application probability is
;       the probability a node will be interior, rather than a leaf.   190
;       Such a tree will have at most 2^md nodes.

(defun rantree (p md)
  (if (or (> (* (random 1000) 0.001) p) (zerop md))
      (combnum (random 3))
      (cons (rantree p (1- md)) (rantree p (1- md)))))

;       (initran n)
;       Initializes random number generator by calling
;       it n times.  When executed in a fresh LISP environment,    200
;       this allows repeatable results.

(defun initran (n) (dotimes (k (1+ n)) (random 1)) 'done)

;       Print control variables:
;       Setting pr causes printing of the tree after each reduction.
;       Setting dpr causes additional, extensive debugging information
;       to be printed.
;       Setting plot-size causes tree size to be written to plot-file
;       after each step.                                            210

(setq pr nil)
(setq dpr nil)
(setq plot-size nil)

;       (limit-multirun seed n md pf limit)
```

```
;       Multiple run facility, with a limit on number of steps.
;       Generates n trees randomly (seeded by sd), given maximum depth md.
;       (The nonleaf probability is fixed at 0.9.)
;       While running displays (R A P) on console, where R is run number,
;       A is tree atom count and and P is number of passes.
;       Writes atom-count / passes summary to a file called
;       pf.
;       Example:  (limit-multirun 37 100 5 "plot" 100)

(defun limit-multirun (seed n md pf limit)
  (setq plot-file (open pf :direction :output))
  (initran seed)
  (dotimes (k n)
           (let* ((rt (rantree 0.9 md))                              230
                  (len (atoms (flatten rt))))
                 (limit-run rt limit)
                 (print (list k len (1- passes)))
                 (print len plot-file)
                 (princ "     " plot-file)
                 (princ (1- passes) plot-file)
                 ))
  (close plot-file)
  'done
  )                                                                  240

;       (size-multirun seed n md pf limit)
;       Multiple run facility with plotting of running
;       tree size (atom count) and limit on number of steps.
;       It keeps track of the sizes during reduction by using
;       size-annotated trees.
;       Generates n trees randomly (seeded by sd), given maximum
;       depth md.  (The nonleaf probability is fixed at 0.9.)
;       After each run, prints tree number, number of passes
;       and final tree size to console.                            250
;       Writes step number and tree size to plot file;
;       data from each tree is separated by a blank line.

(defun size-multirun (seed n md pf limit)
```

```
  (setq plot-file (open pf :direction :output))
  (setq plot-size t)
  (princ "#params " plot-file)
  (prin1 (list seed n md pf limit) plot-file)
  (initran seed)
  (dotimes (k n)                                                    260
           (let ((rt (rantree 0.9 md)))
                (limit-run rt limit)
                (print (list 'tree k
                             'had (1- passes) 'passes
                             'size (size (car root))))
                )
           (terpri plot-file)
           )
  (close plot-file)
  'done                                                             270
  )

;       (atoms tree)
;       Counts the number of atoms in a tree.

(defun atoms (tree)
  (cond
   ((null tree) 0)
   ((atom tree) 1)
   (t (+ (atoms (car tree)) (atoms (cdr tree))))                    280
   ))

;       (size-annotate tree)
;       Adds size annotations to each node of a binary tree.

(defun size-annotate (tree)
  (if (atom tree)
      (cons 1 tree)
      (let ((left (size-annotate (car tree)))
            (right (size-annotate (cdr tree)))                      290
            )
           (cons (+ (size left) (size right))
```

```
                (cons left right))
            )))

;       (untag atree)
;       Removes size-tag from an annotated tree.

(defun untag (L) (cdr L))

;       (tagcar atree)
;       (tagcdr atree)
;       Performs car or cdr on an annotated binary tree.

(defun tagcar (L) (cadr L))
(defun tagcdr (L) (cddr L))

;       (size atree)
;       Return size-tag of an annotated tree.

(defun size (L) (car L))
```

D LISP Definitions for S-Augmentation

These are the additional LISP definitions used for the S-augmentation experiments.

```
;       String−Lambda
;       Routines for running S−augmentation experiments
;       on random SKI−combinator trees.
;       Requires prior loading of SKI−soup2 or SKI−size2
;       for interpreting combinator trees.
;       Bruce MacLennan
;       Collegium Budapest
;       July 1997

;       (one−string−lambda string limit filename)                    10
;       Runs S−augmentation on a given string
;       and writes plot data to filename.
;       Example:  (one−string−lambda (ran−KI−string 50) 150 "plot")

(defun one−string−lambda (string limit filename)
  (open−plot filename)
  (string−lambda string limit)
  (close plot−file))

;       (string−lambda string limit)                                 20
;       Runs a given string of K's and/or I's through
;       random S−augmentation from 0 to 100%.
;       The global variable rstring contains the string
;       being processed (all S's when string−lambda completes).
;       The global variable sites has the indices
;       of non−KI combinators (it's null when
;       string−lambda completes).

(defun string−lambda (string limit)
  (setq rstring string)                                              30
  (let ((n (length string)))
       (setq sites (interval 0 n))
       (dotimes (k n)
                (one−string k n limit)
```

```
                (let ((s (random (- n k))))
                     (setq rstring (replace-site (elt sites s) rstring))
                     (setq sites (delete-site s sites))
                     ))
        (one-string n n limit)
        ))                                                          40

;       (one-string k n limit)
;       Processes the string in rstring, k of whose
;       n elements are S.  At most limit steps are
;       allowed.
;       Write one line to plot file showing percentage
;       of S's and number of steps.

(defun one-string (k n limit)
  (limit-run (bintree rstring) limit)                               50
  (print (list k
               (if no-reduction 'stopped 'terminated)
               'after (1- passes) 'steps))
  (let
   ((size (if plot-size (atoms root) 0)))
   (if plot-size (print (list 'final 'size size)))
   (plot (/ (* 1.0 k) n)
         (1- passes)
         size)
   ))                                                               60

;       (delete-site k string)
;       Deletes item k (indexed from 0) from string.

(defun delete-site (k s)
  (if (zerop k)
      (cdr s)
      (cons (car s) (delete-site (1- k) (cdr s)))
      ))
                                                                    70
;       (replace-site k string)
;       Replaces site k (indexed from 0) of string with S.
```

```
(defun replace-site (k s)
  (if (zerop k)
      (cons 'S (cdr s))
      (cons (car s) (replace-site (1- k) (cdr s)))
      ))

;       (interval s n)                                              80
;       Creates a string of n consecutive integers starting with s.

(defun interval (s n)
  (if (zerop n)
      nil
      (cons s (interval (1+ s) (1- n)))
      ))

;       (ran-KI-string n)
;       Generates a random string of K's and I's of length n.       90

(defun ran-KI-string (n)
  (if (zerop n)
      nil
      (cons (if (zerop (random 2)) 'I 'K)
            (ran-KI-string (1- n)))
      ))

;       (comstring c n)
;       Generates a combinator string of length n                   100
;       composed entirely of c's.

(defun comstring (c n)
  (if (zerop n)
      nil
      (cons c (comstring c (1- n)))))

;       (open-plot filename)
;       Opens a plot files called filename.
                                                                    110
```

```
(defun open-plot (filename)
  (setq plot-file (open filename :direction :output))
  'opened)

;       (plot lambda steps size)
;       Writes one line of plot file containing
;       a lambda value and (optionally) a decremented step count
;       and/or a final tree size. Plot content is determined
;       by global variables plot-steps and plot-size.
                                                                    120
(defun plot (lamb passes size)
  (print lamb plot-file)
  (cond
   (plot-steps
    (princ "     " plot-file)
    (princ (1- passes) plot-file)))
  (cond
   (plot-size
    (princ "     " plot-file)
    (princ size plot-file)))                                        130
  )

(setq plot-steps t)
(setq plot-size nil)
```