The Structure of the Collatz Graph; A Recursive Production of the Predecessor Tree; Proof of the Collatz 3x+1 Conjecture

Kenneth Conrow, Kansas State University (retired)
2371 Grandview Terrace,
Manhattan,
KS 66502
USA

email:kconrow@ksu.edu
August 31, 2010

Abstract

Several major steps develop a proof of the Collatz conjecture. (1)The overall structure of the Collatz graph is a binary predecessor tree whose nodes contain residue sets, $\{c[d]\}$. (2)The residue sets are deployed as extensions (right descents) and l.d.a.s (left descents). (3)Generation of a large initial sample of the tree was accomplished using a recursive computer program. (4)The program output was sorted to show that residue sets which head the l.d.a.s and share a common modulus (the d of $\{c[d]\}$) have multiplicities precisely equal to the successive elements of the Fibonacci series. (5)The density of integers in residue sets provides a measure of the contents of each set, hence that of the whole tree by summation. (6)Use of the established infinite summation, $\sum_1^\infty F_i/2^{i+1} = 1$, shows that all positive integers are accounted for, thus effecting a proof of the Collatz conjecture.

The Collatz problem, perhaps most widely known as the $3x + 1$ problem, is still being actively pursued [9]. It is yet to be proven that any positive integer, when subjected to the Collatz iteration, reaches 1. The usual formula for the Collatz trajectory produces a successor trajectory element T_s from its predecessor T_p.

$$T_s = \begin{cases} (3 * T_p + 1)/2 & \text{when } T_p \text{ odd} \\ T_p/2 & \text{when } T_p \text{ even} \end{cases} \quad \text{(Eq. 1)}$$

To support the approach developed here, the Collatz iteration is better expressed in a combined formula:

$$T_s{=}(3 * T_p + 1)/2^i \text{ where } i \text{ is chosen to give an odd integer } T_s \quad \text{(Eq. 2)}$$

The result is that only odd numbers are shown in the Collatz itineraries. The even integers may be ignored temporarily since they are simply powers-of-two multiples of the odd ones.

The use of varieties of trees as data structures is common in computer applications [8]. To depict the Collatz graph this work employs predecessor trees, in which successors appear above their predecessor elements which are arrayed below them. Assuming the Collatz conjecture, tracing through an integer-by-integer predecessor tree from any point upwards to the root will produce an entire Collatz trajectory.

Two principal features appear in integer-by-integer representations of the predecessor tree, whether as a general tree (p. 3) or as a binary tree (p. 4), i.e. left descent assemblies (l.d.a.s) and extensions. Overall, the integer-by-integer Collatz tree consists of left descent assemblies linked together by extensions.

Left descent assemblies (l.d.a.s) appear in the binary tree as left descents and in the general tree as the sequence of leftmost (smallest numerically) children as the tree is developed layer by layer until a leaf node is reached.

EXAMPLES: Some early l.d.a.s written in predecessor order are 13, 17, 11, 7, 9; 53, 35, 23, 15; 85, 113, 75; and 69.

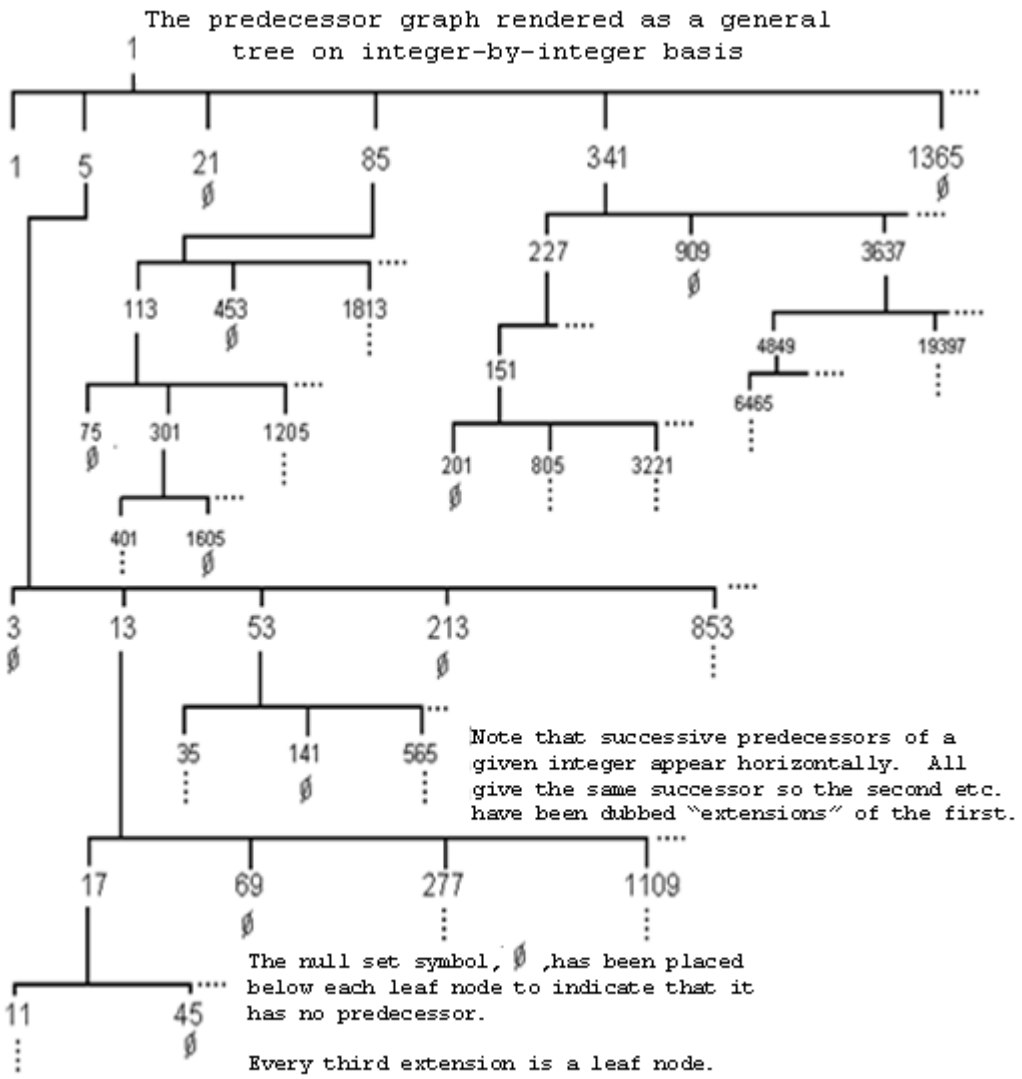
Extensions appear as right descents in the binary tree and as the successive children of a common parent in the general tree representation. An extension element appears as a right neighbor of every node in the tree, each obtained from its left neighbor by use of a recursively applied $4n+1$ relationship. All the members of a set of extensions give the same integer when the Collatz iteration is applied (Eq. 1). The series of extensions of any given integer is infinite.

EXAMPLES: Integer-wise, the extensions of 5: 21, 85, 341, 1365, 5461, ... all give 1 as their successor. The extensions of 3: 13, 53, 213, 853, 3413, ... all give 5 as their successor. The extensions of 113: 453, 1813, 7253, 29013, ... all give 85 as their successor.
Note that the i of Eq. 2 has successive even values in the first and third examples and successive odd values in the second.

Other than using this nomenclature, little further use will be made of trees containing individual integers.

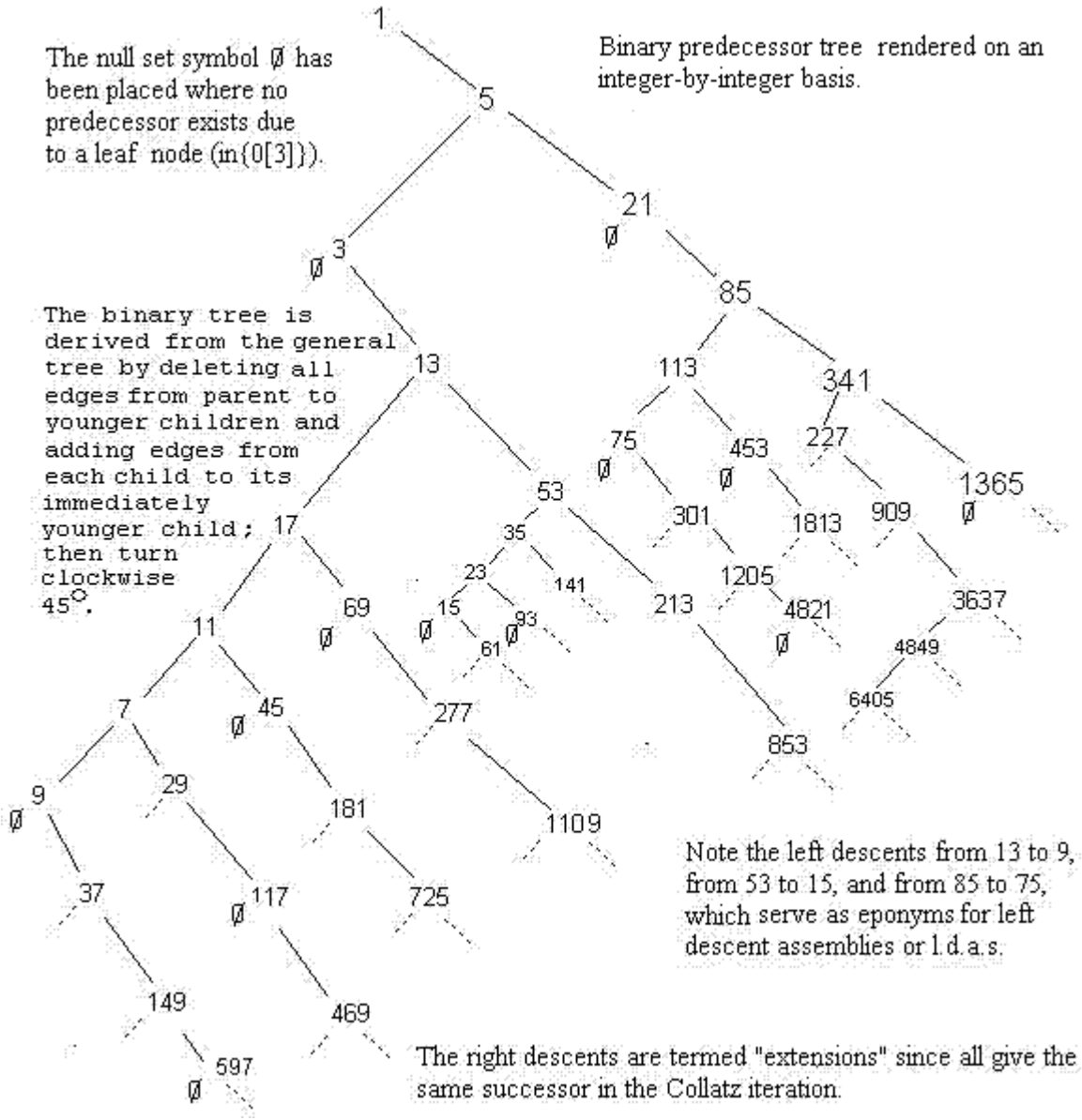
An insightful view is achieved when the predecessor tree, now to be called the abstract tree, is regarded as consisting of residue sets. The notation $c[d]$ indicates an integer congruent to c modulo d, and $\{c[d]\}$ (d even and c odd) indicates an infinite set of such integers [7].

****


The predecessor graph rendered as a general tree on integer-by-integer basis
1
1 5 21 85 341 1365
227 909 3637
113 453 1813
151
4849 19397
6465
75 301 1205
201 805 3221
401 1605
3 13 53 213 853
35 141 565
17 69 277 1109
11 45
Note that successive predecessors of a given integer appear horizontally. All give the same successor so the second etc. have been dubbed "extensions" of the first.
The null set symbol, ∅ ,has been placed below each leaf node to indicate that it has no predecessor.
Every third extension is a leaf node.

****

****



****

The header element of an l.d.a. ($\in \{5[8]\}$), all internal elements, and its leaf node ($\in \{0[3]\}$) are considered as a single assembly. In the case of residue sets which are both header and leaf ($\in \{5[8]\} \cap \{0[3]\}$ i.e. $\{21[24]\}$), the null l.d.a.s consist of a single node. The calculation of predecessors in l.d.a.s proceeds from:

$$T_p = (-1 + \{T_s\} * 2^i)/3 \qquad \begin{cases} \text{where } i = 2 & \text{if } \{T_s\} \in \{1[3]\} \\ \text{or } i = 1 & \text{if } \{T_s\} \in \{2[3]\} \end{cases} \quad \text{(Eq. 3)}$$

thus limiting i to values of $\{1, 2\}$ within l.d.a.s. Each extension is an instantiation of a residue set.

The resulting abstract tree represents paths from individual residue sets not to the root, but rather to an l.d.a. header. A Collatz path can be followed in the abstract tree by tracing any l.d.a. to its header, then identifying the parent that extension came from, and tracing that l.d.a. to its header in turn. The process is repeated until, at last, a member of $\{5[8]\}$ which provides the arrival at 1 is reached. Since the parent of the extension may appear at any position within its l.d.a., parental l.d.a.s may be only partially traversed in this process.

****

Iterative Development of the Abstract Predecessor Tree

Any residue set, {c[3]}, consists of 3 equal subsets:

{0[3]}	{2[3]}	{1[3]}
leaves	s-step	b-step
	↓	↓
	integers equally in	integers equally in
	{0[3]} {1[3]} {2[3]}	{0[3]} {1[3]} {2[3]}

Iterating that process indefinitely yields the entire abstract predecessor tree.

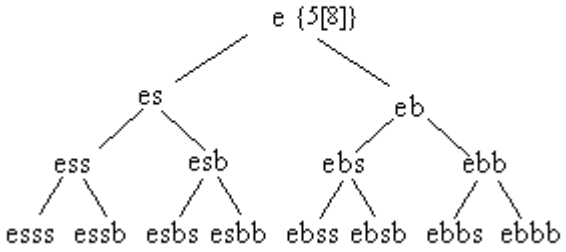
****

It is necessary to distinguish between the two child branches in elaboration of the abstract binary tree according to the value of i in the parent (Eq. 3).

The values of i are limited to 1 or 2 in the predecessor formula to create l.d.a.s. When i is 1, the predecessor value is roughly 4/3 of the successor making it bigger than the parent (a b- step). When i is 2, the predecessor value is roughly 2/3 that of the parent, making it smaller than the parent (an s- step). When $i > 2$, an extension results (Eq. 2).

****

First Three Levels of Abstract Predecessor Tree
Using Symbolic Set Names



****

An l.d.a. is composed of a series of steps, which may be represented by a string "$e(s|b)_j t$" where e represents the header extension, the s and b may appear in any order, and j is the number of steps in the l.d.a. This notation [6] is valuable to completely describe any l.d.a., however lengthy it may be. An l.d.a. is just a limited version of a sequence vector[1]. This notation [6] is valuable to completely describe any l.d.a., however lengthy it may be.

When this expression represents an l.d.a. closed with a leaf node, its termination is marked with a t. When the l.d.a. is not closed, whence l.d.a. development will continue, no terminal t is used.

EXAMPLES: Samples of completed l.d.a.s for comparison between the integer and the residue set contents are:

$13, 17, 11, 7, 9(ebssbt)$	$5[8]_1, 17[32], 11[64], 7[128], 9[512](ebssbt)$
$53, 35, 23, 15(essst)$	$5[8]_6, 3[8]_4, 23[32], 15[64](essst)$
$85, 113, 75(ebst)$	$5[8]_{10}, 17[32]_3, 11[64]_1(ebst)$
$69(et)$	$5[8]_8, (et)$

Early in the tree, the c-values are identical with the integer values because they appear in the initial instantiation of their l.d.a.s, but later the integers are those of higher instantiations as indicated by the subscripts (zero origin indexing).

A residue set provides a measure of the density of the integers it contains. Its density does not change within higher instantiations as the extensions continue infinitely. The measure of the density of the odd numbers in a residue set $\{c[d]\}$ is $1/2d$. Use of $1/d$ would include the tally of the 2^k multiples each odd integer in $\{c[d]\}$ brings with it, since $\sum_{k=0}^{\infty} 1/2^k d = 1/d$. Summation over the densities of the infinity of non-overlapping residue sets should account for all integers in

the abstract predecessor tree. Thus, it would be expected that the summation will yield 1.

Keeping these definitions and conventions in mind, a description is given of the Maple [3] program which builds the abstract tree. Two short examples worked out by hand (p.8-9) illustrate the up and down processing which the program performs.

Programs were written to test and substantiate the structure of the predecessor graph. Each program listing appears in the format and with the line numbers supplied by Maple in its debugging mode. The three components of the *rsetidea* program are *rsetidea* itself, *rsetcurs*, and *densout*. The data structure employed is a stack *mylist* of lists whose elements are detailed in line 3 of $rsetcurs$. -0.4in

```
rsetidea := proc(mxl, mxn) global c, d, lda, mylist, outfile, mxlg, mxng;
   1   outfile := fopen(‘\\maplev4\\mywork\\collatz\\rsetidea.lst‘,WRITE);
   2   mxlg := mxl;
   3   mxng := mxn;
   4   lda := 1;
   5   c := 3;
   6   d := 4;
   7   fprintf(outfile,"%20s %11s %14s %23s %10s %11s \n",
   "modified lda","cdown","ddown","lda at leaf","cup","dup");
   8   mylist := [];
   9   rsetcurs("e",5,8,1);
  10   fclose("\\maplev4\\mywork\\collatz\\rsetidea.lst")
end proc
```

Rsetidea opens the output file, produces its column headers and (much later) closes it, sets up constants for easy reference to individual list elements in the *mylist* stack, and makes the initializing call to *rsetcurs* with the set of arguments which establish the {5[8]} residue set as the root of the abstract tree. The result of the whole program is the production of a complete list of the $\{c[d]\}$ residue sets for every element of every l.d.a. encountered up to the level- and numeric- limits set by the arguments mxl and mxn.

****

First of Two Steps in Derivation of c[d] Values for Sets Within L.d.a.s

The table entries are developed only through three levels of growth, corresponding to the depth of the tree on p.6. The prefixed e is included in the set names to indicate that the envisioned root is a member of the residue set of parents of left descents. Each dn+c formula denotes the set of odd integers congruent to c mod d (i.e. c[d]}).

From	Residue Set	Subset Chosen	Applicable Transform	Next Stage Residue Set	Resulting Set Name	Terminal Set Name
root				8n+5		
e	24n+5	s	(2(24n+5)-1)/3	16n+3	es	
	24n+13	b	(4(24n+13)-1)/3	32n+17	eb	
	24n+21	leaf				et
es	48n+3	leaf				est
	48n+19	b	(4(48n+19)-1)/3	64n+25	esb	
	48n+35	s	(2(48n+35)-1)/3	32n+23	ess	
eb	96n+17	s	(2(96n+17)-1)/3	64n+11	ebs	
	96n+49	b	(4(96n+49)-1)/3	128n+65	ebb	
	96n+81	leaf				ebt
esb	192n+25	b	(4(192n+25)-1)/3	768n+33	esbb	
	192n+89	s	(2(192n+89)-1)/3	384n+59	esbs	
	192n+153	leaf				esbt
ess	96n+23	s	(2(96n+23)-1)/3	64n+15	esss	
	96n+55	b	(4(96n+55)-1)/3	128n+73	essb	
	96n+87	leaf				esst
ebs	192n+11	s	(2(192n+11)-1)/3	128n+7	ebss	
	192n+75	leaf				ebst
	192n+139	b	(4(192n+139)-1)/3	256n+185	ebsb	
ebb	384n+65	s	(2(384n+65)-1)/3	256n+43	ebbs	
	384n+193	b	(4(384n+193)-1)/3	512n+257	ebbb	
	384n+321	leaf				ebbt

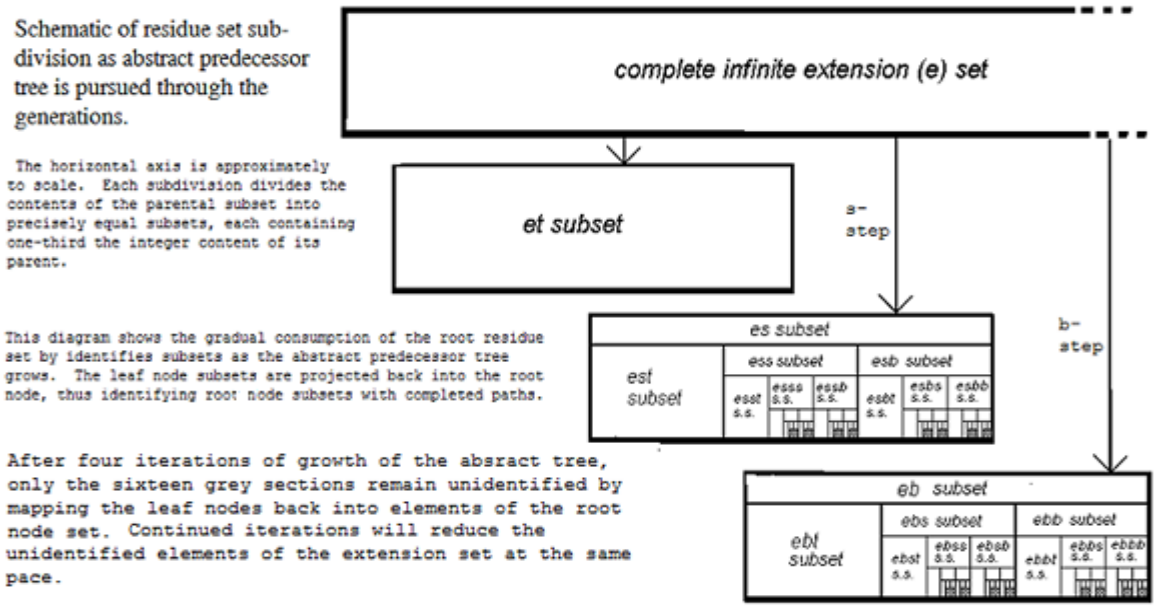
Note: 8n+5 is the root node of the whole tree. It is subdivided into its three subsets, which are congruent to 0, 1, or 2 mod 3. These subsets, in their turn, are similarly subdivided, with the non-leaf subsets extended using the appropriate branch of the predecessor formula. The "$es|bt$" notation shows the progressive development of the contents of the nodes in the abstract tree.

****

```
rsetcurs := proc(path::string, cvalue, dvalue, levelp2)
global mylist, mxng, mxlg;
   1   if mxlg < nops(mylist) or mxng < cvalue then
   2     RETURN(NULL)
       end if;
   3   mylist := [op(mylist), [path, levelp2, cvalue, dvalue]];
   4   if `mod`(cvalue,3) = 0 then
   5     densout(path,cvalue,3*dvalue)
       elif `mod`(cvalue,3) = 1 then
   6     rsetcurs(cat(path,"b"),4/3*cvalue-1/3,4*dvalue,levelp2+2)
       else
   7     rsetcurs(cat(path,"s"),2/3*cvalue-1/3,2*dvalue,levelp2+1)
       end if;
   8   if `mod`(cvalue+dvalue,3) = 0 then
   9     densout(path,cvalue+dvalue,3*dvalue)
       elif `mod`(cvalue+dvalue,3) = 1 then
  10   rsetcurs(cat(path,"b"),4/3*cvalue+4/3*dvalue-1/3,4*dvalue,levelp2+2)
       else
  11   rsetcurs(cat(path,"s"),2/3*cvalue+2/3*dvalue-1/3,2*dvalue,levelp2+1)
       end if;
  12   if `mod`(cvalue+2*dvalue,3) = 0 then
  13     densout(path,cvalue+2*dvalue,3*dvalue)
         elif `mod`(cvalue+2*dvalue,3) = 1 then
  14   rsetcurs(cat(path,"b"),4/3*cvalue+8/3*dvalue-1/3,4*dvalue,levelp2+2)
       else
  15   rsetcurs(cat(path,"s"),2/3*cvalue+4/3*dvalue-1/3,2*dvalue,levelp2+1)
       end if;
  16   mylist := subsop(nops(mylist) = NULL,mylist);
  17   RETURN(NULL)
end proc
```

Rsetcurs is the recursive routine which does the depth-first construction [4] of the abstract tree. Each recursive call builds one more level in a single developing l.d.a. in the abstract tree. Repeated backtracking followed by subsequent calls produce every yet-unexplored branch of the tree, ultimately constructing the complete tree structure. Each level treats all three cases, a leaf node, an s-step and a b- step, in some order. One possibility results from each of the three c-values, c, $c+d$, and $c+2d$ when *rsetcurs* is called with arguments developed from the current parental residue set, {c[d]}. The result is that the density of integers in each level of recursion is 1/3 that of the parent, as illustrated in the accompanying graphic.

****



****

Leaf nodes are encountered exactly once in every invocation of *rsetcurs*, thus assuring completion of l.d.a.s of every possible constitution. A leaf node stops the recursive descent for this residue set without further developing the tree, but *c*- and *d*- values for the leaf node are provided for output by *densout*.

For non-leaf nodes, the next level's *c*-value accompanied by a *d*- value, either multiplied by 2 (for an *s*-step) or by 4 (for a *b*-step), are passed in a recursive call for further development of the tree. The program structure is such that the generation of an infinite binary tree is the only possible result[2]. Of course, limits of time, space, and numeric magnitude prevent complete construction of the infinite abstract tree.

In greater detail, line 1 of *rsetcurs* aborts the depth-first work when the tree has grown beyond the set depth- or numeric magnitude- limits. Each predecessor residue set is tested for a leaf node in lines 4, 8, and 12, for *b*-step parenthood in lines 5a, 9a, and 13a, and finally for *s*-step parenthood in lines 6a, 10a, and 14a. The predecessor formula (Eq. 3) is applied in lines 6-7, 10-11, and 13-14. Whenever a leaf node is discovered, it terminates that

particular l.d.a., and *densout* is called from lines 5, 9, or 13. Each of the non-leaf nodes is given its opportunity for further growth of the incomplete l.d.a. by another recursive call to *rsetcurs*. The parameters passed to the next recursive invocation represent the augmented path with its updated "$e(s|b)_j$" denotation and the {c[d]} value to be parental to the next element of the l.d.a.

```
densout := proc(path::string, tcvalue, tdvalue)
    local step, llevel, dllevel, lcvalue, ldvalue;
    global mylist, lda, c, d, outfile;
 #need local variables starting at passed in value
   1   llevel := nops(mylist);
   2   lcvalue := mylist[llevel,c];
   3   ldvalue := mylist[llevel,d];
   4   fprintf(outfile,"%20s %11d %12d %21s %12d %12d \n",
        cat(path,"t"),mylist[llevel,c],mylist[llevel,d],
        mylist[nops(mylist),lda],tcvalue,tdvalue);
   5   for dllevel from llevel-1 by -1 to 1 do
   6     step := substring(path,dllevel+1 .. dllevel+1);
   7     if step = "b" then
   8       lcvalue := 3/4*lcvalue+1/4;
   9       ldvalue := 3/4*ldvalue
         elif step = "s" then
  10       lcvalue := 3/2*lcvalue+1/2;
  11       ldvalue := 3/2*ldvalue
         end if;
  12     fprintf(outfile,"%20s %11d %12d %21s %12d %12d\n",
          mylist[dllevel,lda],mylist[dllevel,c],mylist[dllevel,d],
          mylist[nops(mylist),lda],lcvalue,ldvalue)
       end do
end proc
```

Densout is called upon completion of every l.d.a. at its leaf node. Line 4 outputs the leaf node both as a terminated node and as a growing l.d.a. node. Lines 5-12 traverse the internal nodes of the completed l.d.a. upward (i.e. in the Collatz direction) from the leaf node taking the successor subsets and outputting their $c[d]$ as it goes along. It is unfortunate that this output is upside down from the general convention used here.

And finally a typical invocation of the whole program:

```
rsetidea(8,1000000);
```

A brief sample of the output from the program in a table, below, demonstrates the development of the abstract tree in depth-first order. Predecessor residue sets are discovered in an order determined by whether c of the parental set is in $\{1[3]\}$ or $\{2[3]\}$, causing a varying order of use of c, $c+d$, and $c+2d$ as the effective parents in the elaborating steps in statements 4-15 of *rsetcurs*. The child residue sets encountered on the way down have been subdivided on the way up so that each of the latter contains a subset of the former. This identifies each subdivision of the parental residue sets, each an element in a particular l.d.a. to its own deeper leaf node set. The expected 1/3 decrease of the density from parental to child residue set causes the ratios of d_{up} to d_{down} to decrease as $1/3^k$ as deeper residue sets are developed (as indicated parenthetically in the table). The d values reflect a multiplication by 2 or 4 according to whether s- or b- steps were taken.

The first few lines of output (below) reveal an extensive duplication of integers in the calculated residue sets. The side-by-side presentation of 2 $c[d]$ values provided by *densout* provides a basis for understanding the integer duplication in the tree's residue sets. The first (in $c_{down}[d_{down}]$) gives the denser residue set obtained higher in the tree, and the second (in $c_{up}[d_{up}]$) gives the specific residue sets on the narrow path to the leaf node just located.

modified lda	cdown	ddown	lda at leaf	cup	dup	(ddown/dup)
est	3	16	es	3	48	
e	5	8	es	5	24	
esbbt	33	256	esbb	33	768	
esb	25	64	esbb	25	192	* 1/3
es	3	16	esbb	19	144	* 1/9
e	5	8	esbb	29	216	* 1/27
esbbbbt	513	4096	esbbbb	513	12288	
esbbb	385	1024	esbbbb	385	3072	* 1/3
esbb	33	256	esbbbb	289	2304	* 1/9
esb	25	64	esbbbb	217	1728	* 1/27
es	3	16	esbbbb	163	1296	* 1/81
e	5	8	esbbbb	245	1944	* 1/243

The lines marked with * call attention to cases where the up residue set is a subset of the down residue set. Hence, care must be taken in the projected summation of the densities of the produced residue sets to avoid over-counting the total density of integers.

The source of the duplication is that predecessor residue sets reached by lines 4-7 of *rsetcurs* (with no d value added into c) produce child sets whose c value is identical to those of the parents. This occurs very frequently in the early output because the case of passing c without any d added is earliest in the code. If the density of integers in these children were included in the overall density in the abstract predecessor tree, the total integer density would be inflated. The children produced by *rsetcurs* by the branches from statements 8-11 (with d added in) and 12-15 (with $2d$ added in) produce headers of l.d.a.s with new c values, so no such duplication of integer content occurs there. The entire density of the headers produced by *rsetcurs* in lines 4-5 is accounted just as any other node is, including the densities which will arise in its immediate children. Therefore, it is sufficient to sum the density of integers in only the l.d.a. header nodes when determining the total integer content.

Several features indicate the correct performance of the *rsetidea* program and are consistent with production of an infinite abstract tree. All possible permutations of s- and b- steps within l.d.a.s of the same composition are exhibited (indicating the search is complete). The c values in the l.d.a. headers are all different from one another (indicating uniqueness throughout each specific l.d.a.). The d values within l.d.a.s exhibit progressively higher powers of 2 (indicating the achievement of very large numbers in long l.d.a.s). The fact that the predecessor graph is a tree, providing additionally that it contains all the integers, would prove the Collatz conjecture [9].

A tabulation of the {c[d]} values of the l.d.a. header nodes sorted and separated from all the others illustrates the avoidance of any duplication of integers within these residue sets.

d-value order		
et	5	8
est	3	16
ebt	17	32
esst	23	32
ebst	11	64
essst	15	64
esbt	25	64
ebsst	7	128
esbst	59	128
ebbt	65	128
essbt	73	128
esssst	95	128
esbbt	33	256
esbsst	39	256
ebbst	43	256
essssst	63	256
esssbt	105	256
ebssst	175	256
ebsbt	185	256
essbst	219	256

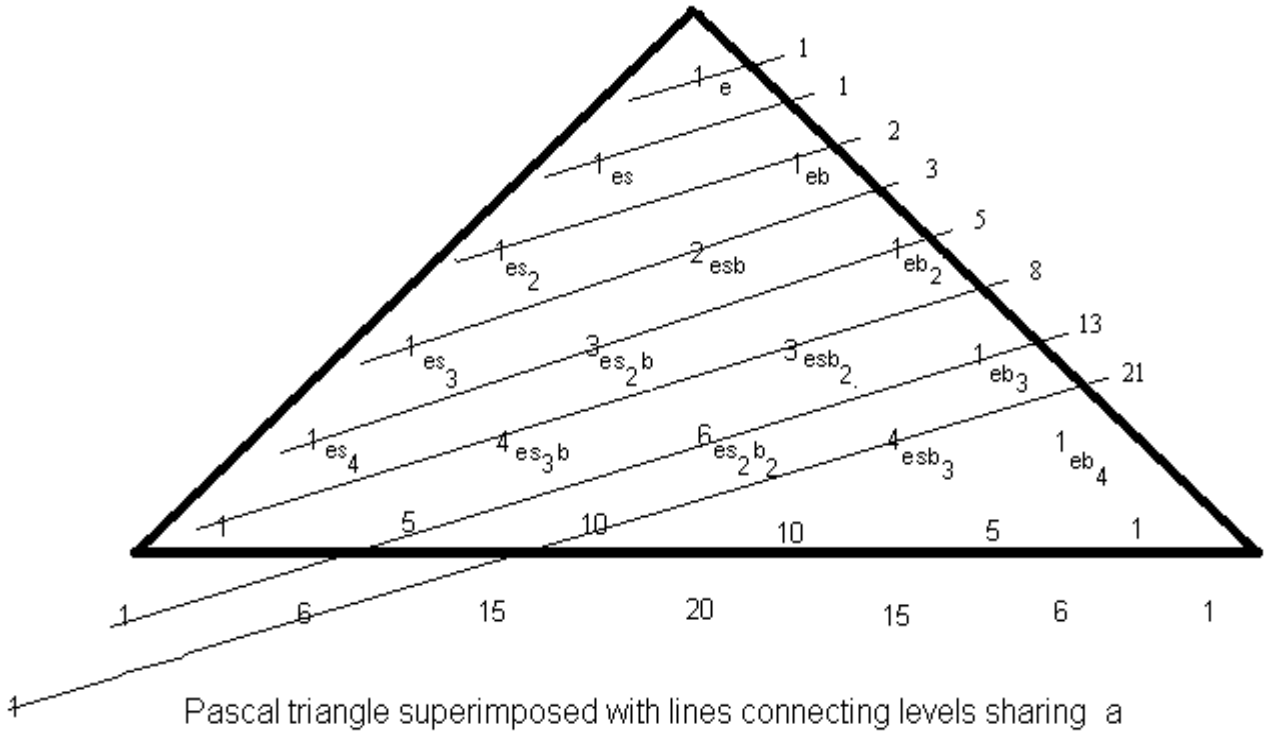
header lda order		
et	5	8
ebt	17	32
est	3	16
ebbt	65	128
ebst	11	64
esbt	25	64
esst	23	32
ebbbt	257	512
ebbst	43	256
ebsbt	185	256
ebsst	7	128
esbbt	33	256
esbst	59	128
essbt	73	128
essst	15	64

In the left table, the residue sets are in order by the header d-value, and c-values within them. Observe the multiplicity of residue sets sharing a given d value. In the right table, the header paths are in order by their lengths and alphabetically within each length.

The table sorted on d- values reveals a means of doing the desired summation of the integer densities in the totality of the residue set headers.

l.d.a.	{c[d]}		2/d	mult	mult*2/d	F(i) term
et	5	8	1/4	1	1/4	1/4
est	3	16	1/8	1	1/8	1/8
ebt	17	32	1/16			
esst	23	32	1/16	2	2/16	2/16
ebst	11	64	1/32			
essst	15	64	1/32			
esbt	25	64	1/32	3	3/32	3/32
ebsst	7	128	1/64			
esbst	59	128	1/64			
ebbt	65	128	1/64			
essbt	73	128	1/64			
esssst	95	128	1/64	5	5/64	5/64

****



Pascal triangle superimposed with lines connecting levels sharing a given number of powers of two in the abstract generation tree. These entries sum to the successive elements of the Fibonacci series.

****

As the above diagram suggests, the multiplicity of entries sharing an identical density (i.e. d-values) goes as the successive elements of the Fibonacci series. The summation of the elements of the Fibonacci series divided by successive powers of 2 is $\sum_1^\infty F_i/2^{i+1}$ which evaluates to 1. [5] In our case the terms in the summation of the Fibonacci series exactly equal the total of the combined densities of the several residue sets which share each particular power of 2 as their d value, so it, too, sums to 1. This indicates that the tree does contain all the positive integers.

Thus, in addition to the inductive proof by construction of the abstract predecessor tree, we have come to a proof by enumeration of the infinity of the integer contents of the abstract predecessor tree.

References

[1] http://groups.google.com/group/sci.math/msg/c1181e6009302585l also http://www-personal.ksu.edu/~kconrow/glossary#sequence.html, Mensanator's sequence vector bears a close correspondence to left descent assemblies. If the *s*- and *b*- symbols of l.d.a.s are replaced with the integers *1* and *2*, respectively, and the Collatz successor order is used, it is seen that l.d.a.s are simply sequence vectors starting at a leaf node, ending with a node in {5[8]}, and containing only the integers 1 and 2 in the vector.

[2] *Binary tree*, http://en.wikipedia.org/wiki/Binary_tree, This article provides background for the term "binary tree".

[3] ©*Maple*, Waterloo Maple Inc., Waterloo, Canada. The *sine qua non* for this work.

[4] *Depth-first search*, http://en.wikipedia.org/wiki/Depth_first_search, This article details one processing potential for the tree as a data structure.

[5] *Fibonacci number*, http://en.wikipedia.org/wiki/Fibonacci_number, Source of the Fibonacci sum expression.

[6] *Formal language*, http://en.wikipedia.org/wiki/Formal_language, The idea of describing l.d.a.s as character strings comes from formal language practices in computer science.

[7] *Modular arithmetic*, http://en.wikipedia.org/wiki/Modular_arithmetic, The concepts of modular arithmetic and residue sets are explained.

[8] *Tree (data structure)*, http://en.wikipedia.org/wiki/Tree_data_structure, This article provides background on the use of trees as data structures in computing.

[9] Jeffrey Lagarias, *The 3x+1 problem and its generalizations*, http://www.cecm.sfu.ca/organics/papers/lagarias/paper/html/paper.html and http://arxiv.org/abs/math.NT/0309224.