TIGRE: Combinator Graph Reduction On The RTX 2000

Philip Koopman, Jr.

Harris Semiconductor
2525A Wexford Run Rd.
Wexford, PA 15090

Abstract

My dissertation work investigated an efficient evaluation technique for lazy functional programs based on combinator graph reduction. Graph reduction is widely believed to be slow and inefficient, but an abstract machine called the Threaded Interpretive Graph Reduction Engine (TIGRE) achieves a substantial speedup over previous reduction techniques. The runtime system of TIGRE is a threaded system that permits self-modifying program execution with compiler-guaranteed safety. This paper describes an implementation of TIGRE in Forth for the Harris RTX 2000 stack processor.

Introduction

The TIGRE (Threaded Interpretive Graph Reduction Engine) abstract machine is a highly efficient mechanism for executing combinators in a pure graph-reduction style **[KOOP89]**, **[KOOP90]**. TIGRE maps efficiently onto conventional hardware, providing speeds that compare quite favorably with previous combinator graph reducers.

Since the interpreter of TIGRE is similar to a Forth threaded code interpreter, Forth is an excellent language for exploring the implementation of TIGRE. In fact, the first implementation of TIGRE was done in Forth. This paper is a description of an implementation of the TIGRE abstract machine on the Harris RTX 2000 stack-based processor.

Background

[TURN79] described a technique for implementing normal-order functional languages, sometimes referred to as SK-combinator reduction. The idea is based on the observation that all of the variables in a functional program can be removed by transforming it into a sequence of combinators which are drawn from a small, pre-defined set of combinators. With all free variables thence removed, the resulting program text becomes amenable to representation as a graph in which subgraph sharing represents the sharing of subexpressions in the program, cycles represent recursion, and combinator definitions represent graph rewrite rules.

As an example, Figure 1 shows the operation of the S combinator. When executed, the S combinator allocates nodes 3 and 4 from a garbage-collected heap, and rewrites the contents of node 0 to transform the subgraph (((S f) g) x) into ((f x)(g x)). Amazingly, only two combinators, **S** and **K**, are necessary to represent any program! (Although in practice, it is desirable to introduce other combinators and integers for the sake of efficiency.) In this scheme, executing programs becomes a process of graph reduction. Unfortunately, previous methods of graph reduction have been extremely inefficient.

TIGRE, A New Architecture for Pure Graph Reduction

The major sources of inefficiency in most graph reducers are the traversing of the graph's left spine (stack unwinding) and the case analysis of node tags. If these costs are reduced or eliminated, significant speedups may be possible.

One of the key points of TIGRE is the elimination of most of the overhead for traversing tree nodes during the stack unwinding process. This can best be accomplished by simply eliminating the need for tags, thereby eliminating the cost of tag interpretation. Figure 2 shows a generalized node representation which has tags associated with both the left and right-hand sides of the node. Figure 3 shows a tree for the expression ((+ 11) 22) which we shall use as a running example. The numbers above the nodes serve as labels for our discussion. Although only three tag types are shown in the example, typically more tag types are used in actual implementations.

As a first step in eliminating tags, we shall replace the cells containing constant values by pointers to indirection nodes. (We are assuming, without loss of generality, that all data items are integers.) Figure 4 shows the result of this rewriting. Any graph can be rewritten with constant values placed in the right-hand sides of indirection nodes in a similar manner.

Now, notice that constants are only found as arguments to indirection combinators. If we rename those **I** combinators in the left-hand side of constant nodes as **LIT** combinators (short for "literal value" combinators), as shown in Figure 5, the constant tag is no longer needed, since the **LIT** combinator implicitly identifies the argument as a constant value. All other special tag types can be eliminated by defining new combinators in a similar manner.

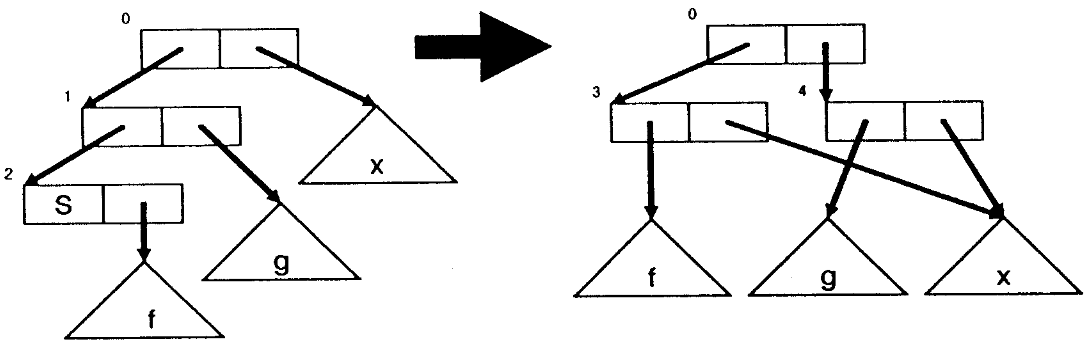

Figure 1. Operation of the S combinator.

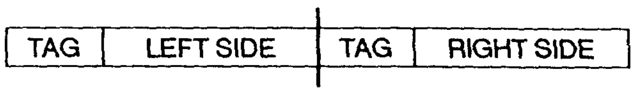

Figure 2. Basic structure of a node.

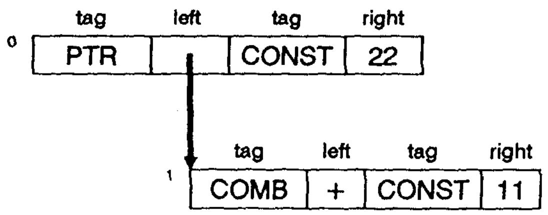

Figure 3. Example for expression ((+ 11) 22).

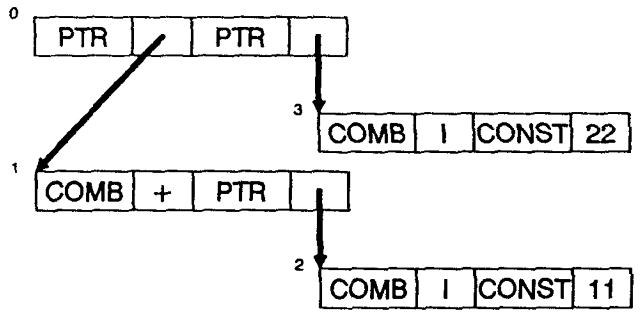

Figure 4. Example using indirection nodes for constants.

The graph shown in Figure 5 now only has two tag types: combinator and pointer. We can now greatly reduce the cost of tag checking by using any number of standard tricks. For instance, all nodes and therefore pointer values can be aligned on 2- or 4-byte boundaries (which improves speed or is even required on many machines). The lowest bit of a cell's contents can then be used as a one-bit tag. Figure 6 shows the graph rewritten in this style.

The case analysis for numeric constants has been replaced by the need to reduce LIT combinators (although we argue that this combinator is often present in the form of an I node anyway). However, we have also reduced the amount of tag checking on all other cells. This is the representation used for the C language implementation of TIGRE.

The Key Insight

The generic graph shown in Figure 7 is executed by traversing the leftmost spine, placing pointers to ancestor nodes onto a spine stack. When a combinator is encountered in the graph, some code to implement the combinator is executed. The data structure is controlling the execution of the program. Another, more insightful, view is that the data structure is itself a program with two instruction types: pointer and combinator. Then graph reduction is essentially a process of interpreting a threaded program that happens to reside in the node heap. In other words, the tree is a program that consists mainly of calls to subroutines. These subroutines then contain calls to other subroutines, and so on until, finally, some other executable code is found. The C implementation of TIGRE thus is actually a threaded code interpreter [BELL73].

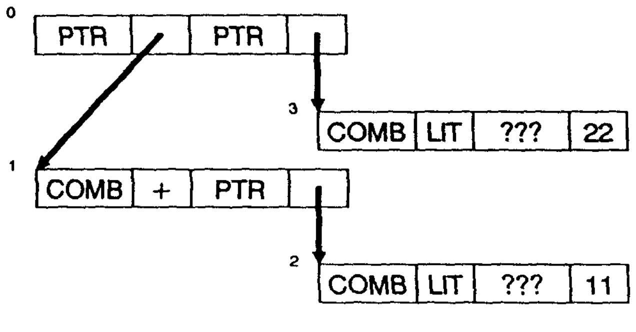

Figure 5. Example using LIT nodes instead of indirection nodes for constants.

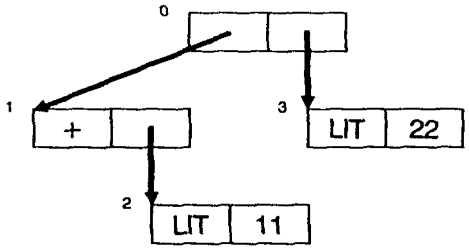

Figure 6. Example with tags removed.

The key idea is that *the spine stack is actually just a subroutine return stack* for the interpreted threaded program. As control flows from node 0 to node 1 to node 2 to node 3 in the graph of Figure 7, pointers to these nodes are stored on the spine stack. These pointers will eventually be used to access the right-hand side values of the ancestor nodes as arguments to a combinator, so what we really want saved on the stack are pointers to the right-hand sides of each node. If the left-hand sides of each node are viewed as subroutine call instructions, then the return addresses which would be automatically saved would be the right-hand cell addresses of the spine of the graph, which is exactly the desired behavior.

Combinator nodes, such as node 3 in Figure 7, contain some sort of token value that invokes a combinator. At some point during program execution, this value will have to be resolved to an address for a piece of code to be executed, so the assembler version of TIGRE simply stores the actual code addresses of the combinator action routines instead of token values. In fact, we store a subroutine call to the combinator code, so the address of the right-hand side of node 3 in Figure 7 will be pushed onto the spine stack, and the combinator will have all its arguments pointed to by the spine stack (which is now the subroutine return stack). A pleasant side effect of this scheme is that there is now only one type of data in the graph: the pointer. Hence there is only one type of node, and therefore *no conditional branching or case analysis is required at runtime*. All nodes contain either pointers to other nodes or pointers to combinator code. Since all node values (except the right-hand sides of LIT cells) are subroutine call instructions, we can simplify matters by simply saying that each cell contains a pointer that is interpreted as a subroutine call by the TIGRE execution engine.

In a Forth implementation, TIGRE graph nodes can be implemented as pairs of subroutine call words. Since the RTX 2000 supports direct execution of a graph pointer as a subroutine call

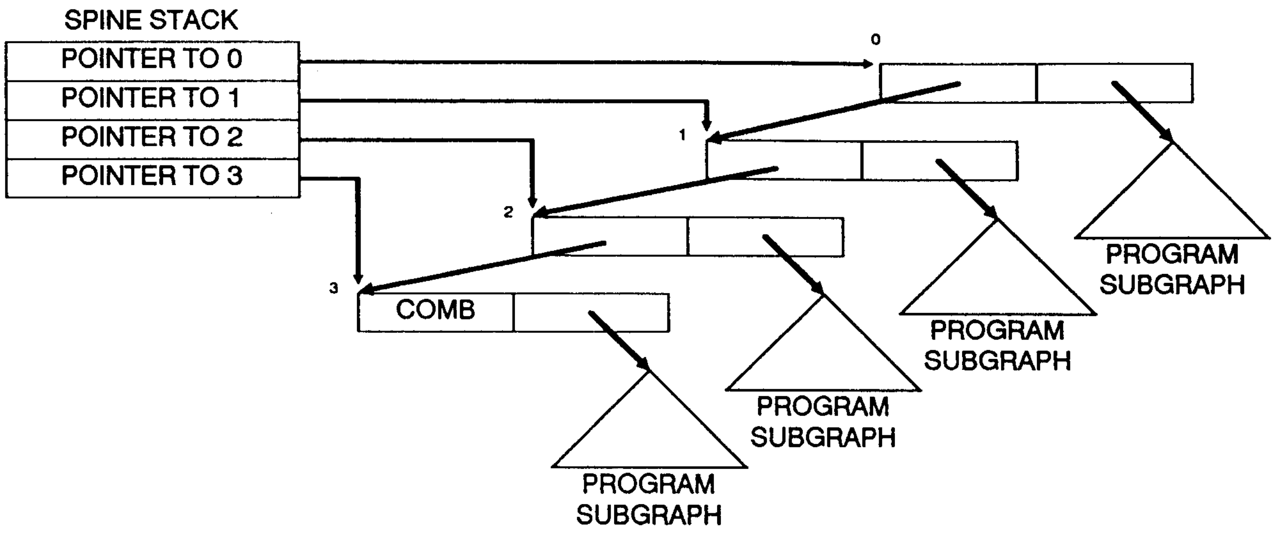

Figure 7. An example TIGRE program graph, emphasizing the left spine.

instruction, the hardware's native subroutine calling mechanism is used to traverse the spine, using the subroutine return stack as the spine stack. Thus, the representation of the graph shown in Figure 8 is actually a directly executable data structure on the RTX 2000.

While the graph shown in Figure 8 is simple, its operation is not necessarily obvious. Evaluation of a program graph is initiated by doing a subroutine call to the left-hand side of the root node of a subgraph. The machine's program counter then traverses the left spine of the graph structure by executing the call instructions of the nodes following the leftmost spine. When a node points to a combinator, the RTX 2000 simply begins executing the combinator code, with the return address stack providing addresses of the right-hand sides of parent nodes for the combinator argument values.

The processor is in no sense interpreting the graph. It is *directly executing* the data structure, using the hardware-provided subroutine call instructions to do the stack unwinding. The data structure is modified every time a combinator is executed, resulting in a situation where the processor is executing self-modifying code. However, unlike other self-modifying code techniques, the compiler can completely guarantee the correctness and safety of the process. Indeed, since graph reduction is inherently a self-modifying process, one could say that techniques that do not rely on self-modifying code are by their very nature inefficient.

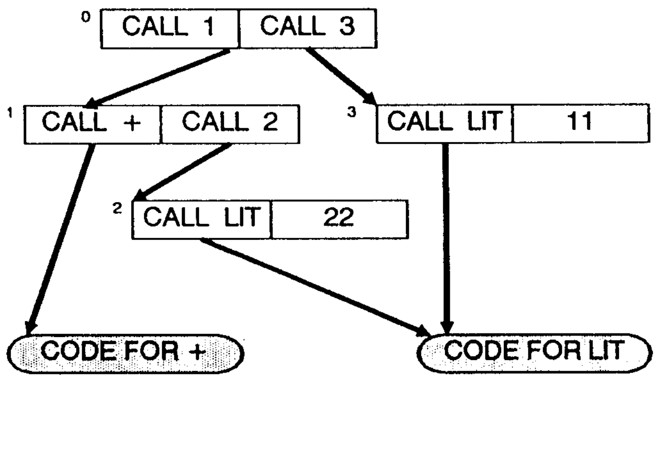

Figure 8. Example with pointers replaced by subroutine call instructions.

Briefly, TIGRE uses an interpretive pointer to do subroutine call operations down the left spine of the graph. When combinators are reached, they pop their arguments from the return stack, perform graph rewrites, and then jump to the new subgraph to continue traversing the new left spine. The use of the return stack for graph reduction is slightly different than for "normal" subroutines in that subroutine returns are never performed on the pointers to the combinator arguments, but rather, the addresses are consumed from the return stack by the combinators. (This seems to be a characteristic of other combinator reducers as well).

An RTX 2000 Implementation

The code listing at the end of this paper shows an implementation of TIGRE for the Harris RTX 2000. The RTX 2000 is a stack-based processor that can efficiently execute Forth programs **[HARR88]**. In particular, the RTX 2000 supports single-cycle subroutine calls, which should make the graph traversal threading operation extremely fast.

In the RTX 2000 implementation, combinators are built as Forth words with braces around the name of the combinator (*e.g.,* **{S}** for the S combinator). The word **^'** is used to compile a reference to a combinator, and the word **^** is used to compile a reference to another heap node. Note that on the RTX 2000 subroutine calls are stored as the call address shifted right one bit, causing a proliferation of **2*** and **U2/** shifting operators through the code. A full-featured implementation would include a stop-and-copy garbage collector; this code aborts when it is out of heap space.

Two examples are included in the listing. The routine **SUC** takes an integer on the data stack, builds a graph to compute the successor function, executes the graph, and returns the result on the data stack. Similarly, **FIB** computes the Nth Fibonacci number. Note that the use of the data stack for parameter passing means that graph reduction code can be seamlessly merged with traditional Forth code, as is done with the word **FIB-TABLE**.

The RTX 2000 version of TIGRE performs approximately 450K reduction applications per second (RAPS) for the SKI combinator set implementation of FIB on a 10 MHz RTX 2000. This is almost twice the speed as the NORMA machine which was custom-built specifically for graph reduction **[SCHE86]**. It is also faster than the 322K RAPS rating of a SparcStation I workstation and the Cray Y-MP at 352K RAPS. In fact, only the 16.67 MHz DECstation 3100, which uses the MIPS R2000 processor to obtain 495K RAPS, is faster than the RTX 2000. Most of this speed advantage for the R2000 comes from its support for a split instruction and data cache, giving it twice the available memory bandwidth available as the RTX 2000 (but, the RTX 2000 is still faster if the system clock speeds are normalized).

One of the important application areas for graph reduction is in parallel processing. In this area, the absolute speed of the processor is not as important as the aggregate speed of the processors that may affordably be placed in a box. Stack processors, with their high level of integration and small transistor count, seem to provide good levels of performance while requiring minimal hardware resources, and therefore may be well suited as processor units in a graph reduction parallel processing system. Of course, 16-bit processors have limited addressability, so a 32-bit stack processor would be more desirable for a large-scale project.

Further Reading

The subject of graph reduction and functional programming languages is not easily grasped upon casual inspection. The following publications are recommended for those interested in learning more about this subject. **[BACK78]** ignited the current interest in functional programming with his Turing Award lecture. **[HEND80]** and **[FIEL88]** are commonly available textbooks on functional programming. **[PEYT87]** contains detailed descriptions of implementation techniques for functional programming languages, including a chapter on combinator graph reduction techniques. **[BELI87]** described an eager graph reduction implementation in Forth. **[HUGH84]** presents a thought-provoking discussion of the importance of functional programming (many of the arguments in this article also apply to Forth, although perhaps in different ways).

Conclusions

Through its simplicity, TIGRE gives fundamentally better insight into the operation of graph reducers. Since TIGRE uses threaded code techniques and stack-based references to the spine stack, Forth makes an excellent environment for experimenting with implementation techniques. The RTX 2000 provides good performance for TIGRE with minimal hardware cost, making it attractive for use as a parallel processing unit for graph reduction.

Acknowledgements

My dissertation research was supported by NASA/Goddard under contract NAG-5-1046, monitored by Dr. John Dorband. Profs. Dan Siewiorek and Peter Lee of Carnegie Mellon University contributed to the development of TIGRE and analysis of its performance. Harris Semiconductor is funding my continued involvement in advanced programming language implementation research. Rick VanNorman helped me get significant speed improvements in the RTX 2000 version of TIGRE.

References

[BACK78] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, **21**(8):613-641, August, 1978.

[BELI87] Belinfante J.G.F., "S/K/ID: Combinators in Forth," *Journal of Forth Application and Research*, **4**(4):555-580, 1987.

[BELL73] Bell, J., "Threaded code," *Communications of the ACM*, **16**(6):370-372, June, 1973.

[FIEL88] Field, A.J. & Harrison, P.G., *Functional Programming*, Addison-Wesley, Wokingham England, 1988.

[HARR88] Harris Semiconductor, *RTX-2000 Real Time Express Microcontroller Data Sheet*, Harris Corporation, Melbourne FL, 1988.

[HEND80] Henderson, P., *Functional Programming*, Prentice-Hall, 1980.

[HUGH84] Hughes, R.J., *Why Functional Programming Matters*, Chalmers University of Technology, Goteborg Sweden, 1984.

[KOOP90] Koopman, P., *An Architecture for Combinator Graph Reduction*, Academic Press, 1990.

[KOOP89] Koopman, P., Lee, P., "A Fresh Look at Combinator Graph Reduction," *Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland OR, June 21-23, SIGPLAN Notices*, **24**(7):110-119, July 1989.

[PEYT87] Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall, London, 1987.

[SCHE86] Scheevel, M., "NORMA: A graph reduction processor," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge Massachusetts*, pp. 212-219, ACM, August, 1986.

[TURN79] Turner, D.A., "A new implementation technique for applicative languages," *Software - Practice and Experience*, **9**(1):31-49, January, 1979.

Source Listing

```
\ TIGRE for the RTX 2000 using AppForth
\ (C) Copyright 1990, all rights reserved - Philip Koopman
\ Special thanks to Rick VanNorman for significant speedups
DECIMAL

\ Portability & trace words
: CELLS  2* ;
: t\  [COMPILE] \ ; immediate   \ Make a nop for run-time trace

\ Heap space management
5000 CONSTANT #HEAP-CELLS        \ Size of heap for tree data
VARIABLE HEAP-SPACE  #HEAP-CELLS  CELLS 2*  100 + ALLOT
HEAP-SPACE  #HEAP-CELLS CELLS 2* +  CONSTANT HEAP-LIMIT

: INIT-HEAP  ( - )
   HEAP-SPACE  4 G!  HEAP-LIMIT 6 G! ;

: HEAP_ALLOCATE   ( nbytes - first.addr )
  4 G@  +  4 DUP_G!
  DUP  6 G@  U>
  IF -1 ABORT" Out of heap space!" THEN ;

\ Initial tree building helpers
: ^   CELLS 2*   HEAP-SPACE +  >ta  U2/ ;
: ^'  ?COMP  '  >TA  U2/ <literal>   ; IMMEDIATE
: h,  SWAP   4 G@  !  1 CELLS   4 G@ + 4 G!
             4 G@  !  1 CELLS   4 G@ + 4 G! ;
```

```
\ Compilation helping words
: ]cells  " CELLS ] literal " evaluate  ; IMMEDIATE
: 2args   " R> @ 2* execute " evaluate ( Evaluate 1st argument )
          " R@ @ 2* execute " evaluate ( Evaluate 2nd argument ) ;
IMMEDIATE
: strict  " [ {lit} ] LITERAL OVER R> 2 !- ! " EVALUATE ; IMMEDIATE

\   I x -> x    ( perform a jump to x )
: {I}   ( - )    ( RS: ^myrhs - )   t\ ."  I "
  R> @  2* >R;  ;

\   K c x -> I c  ( perform a jump to c )
: {K}   ( - )    ( RS: ^parentrhs ^myrhs - )   t\ ."  K "
  R> @  R>DROP  2* >R;  ;

\   S f g x ->  (f x) (g x) ( perform a jump to rhs address )
: {S}   ( - )    ( RS: ^grandrhs ^parentrhs ^myrhs - )   t\ ."  S "
   R> @                       \ f
   R> @                       \ f g
   R@ @ DUP                   \ f g x x
   [ 4 ]cells HEAP_ALLOCATE   \ f g x x a
   [ 3 ]cells +               \ f g x x d
   4 !-                       \ f g x b
   2 !+  DUP>R                \ f g c
   4 !-                       \ f a
   Ø !+  U2/                  \ ^a
   R>    U2/                  \ ^a ^c
   R>                         \ ^a ^c f
   2 !-                       \ ^a e
   Ø !+                       \ e
   >R; ;

$DE2Ø CONSTANT {LIT}

\   + x y -> LIT sum  ( perform a return )
: {+}   ( - )    ( RS: raddr ^parentrhs ^myrhs - )  t\ ."  + "
   2args +  strict ;

: {-}   ( - )    ( RS: raddr ^parentrhs ^myrhs - )  t\ ."  - "
   2args - strict ;

: {<}   ( - )    ( RS: raddr ^parentrhs ^myrhs - )  t\ ." < "
   2args < strict ;

\   IF c x y -> I x || I y
: {IF}   ( - )   ( RS: ^grandrhc ^parentrhs ^myrhs - )  t\ ."  IF "
   R> @ 2* EXECUTE
   IF
      R> @
      ^' {I}
      OVER R>
      2 !-
      !
      2* >R;
   THEN
   R>DROP
   R>
   2 @-
   ^' {I}
   SWAP !
   2* >R; ;

\ Fast constants for small integers
: {1} 1 ;
: {2} 2 ;
: {3} 3 ;

\ EXAMPLE: suc(n) = n+1
\ ((S ((S (K +) ) (K 1) ) ) I ).
: SUC  ( n - suc )
  INIT-HEAP
   ( Ø )  2 ^   1 ^ h,
     ( 1 )  {lit}  SWAP h,
     ( 2 )  3 ^  ^' {I} h,
       ( 3 )  ^' {S}  4 ^ h,
         ( 4 )  5 ^  7 ^ h,
           ( 5 )  ^' {S}  6 ^ h,
             ( 6 )  ^' {K}  ^' {+} h,
           ( 7 )  ^' {K}  8 ^ h,
             ( 8 )  {lit}   1 h,
  2 CELLS HEAP_ALLOCATE DROP HEAP-SPACE >R;  ;
```

```
\ EXAMPLE: computes nth Fibonacci number
\ ((S ((S ((S (K IF)) ((S <)  (K 3)))) (K 1)))
\     ((S ((S (K +)) ((S (K CYCLE)) ((S -) (K 1)))))
\    ((S (K CYCLE)) ((S -) (K 2)))))).
\ Note: CYCLE refers to node 2, which is root of function subtree
: <FIB>  ( n - suc )
  INIT-HEAP
   ( Ø )  2 ^   1 ^ h,
     ( 1 )  {lit}  SWAP h,
     ( 2 )  3 ^   15 ^ h,
       ( 3 )  ^' {S}   4 ^ h,
         ( 4 )  5 ^   13 ^ h,
           ( 5 )  ^' {S}   6 ^ h,
             ( 6 )  7 ^   9 ^ h,
               ( 7 )  ^' {S}   8 ^ h,
                 ( 8 )  ^' {K}   ^' {IF} h,
               ( 9 )  10 ^   11 ^ h,
                 ( 10 )  ^' {S}   ^' {<} h,
                 ( 11 )  ^' {K}  ^' {3} h,
                                      ( 12 )  {lit}   3 h,
           ( 13 )  ^' {K}  ^' {1} h,
                                      ( 14 )  {lit}   1 h,
       ( 15 )  16 ^   27 ^ h,
         ( 16 )  ^' {S}   17 ^ h,
           ( 17 )  18 ^   20 ^ h,
             ( 18 )  ^' {S}   19 ^ h,
               ( 19 )  ^' {K}   ^' {+} h,
             ( 20 )  21 ^   23 ^ h,
               ( 21 )  ^' {S}   22 ^ h,
                 ( 22 )  ^' {K}   2 ^ h,   \ cycle
               ( 23 )  24 ^   25 ^ h,
                 ( 24 )  ^' {S}   ^' {-} h,
                 ( 25 )  ^' {K}   ^' {1} h,
                                      ( 26 )  {lit}   1 h,
         ( 27 )  28 ^   30 ^ h,
           ( 28 )  ^' {S}   29 ^ h,
             ( 29 )  ^' {K}   2 ^ h,   \ cycle
           ( 30 )  31 ^   32 ^ h,
             ( 31 )  ^' {S}   ^' {-} h,
             ( 32 )  ^' {K}   ^' {2} h,
                                      ( 33 )  {lit}   2 h,
  2 CELLS HEAP_ALLOCATE DROP  HEAP-SPACE >R;  ;

: FIB-TABLE  ( - )
  CR  ."  n    FIB(n)"
  CR  ." --    ---" CR
  13 1 DO
      I  3 U.R   I FIB  7 U.R  CR
  LOOP  ;
```