Appendix B
Selected TIGRE Program Listings

This appendix contains abbreviated program listings for TIGRE as implemented in C, VAX assembly language, and MIPS R2000 assembly langauge.

B.1. REDUCE.H

```
/* TIGRE Graph Reducer - - Master header file
        REDUCE.H */
/* (C) Copyright 1989   Philip Koopman Jr.  */
/*  Last update:  4/17/89   */

#define TRUE   1
#define FALSE  0
/* The TURBOC timing routines are primitive
        - - so don't bother to use them */

#ifdef TURBOC
#define UNIX FALSE
#else
#define UNIX TRUE
#endif

#ifdef VAX        /* adapt to VAX assembler
        kernel */
#define VAX_ASM TRUE
#else
#define VAX_ASM FALSE
#endif

#ifdef INST
#define INSTRUMENTS TRUE
#else
#define INSTRUMENTS FALSE
#endif

#define DEBUG  FALSE
#define DEBUG_TRACE FALSE+DEBUG
#define DEBUG_HEAP  DEBUG+FALSE
#define DEBUG_DUMP  FALSE

/* If not UNIX, then operating in TURBOC */

#include <stdio.h>

#if UNIX
#include <sys/types.h>
#include <sys/times.h>

/* time measurement definitions */
#define     HZ   60
#define INT_32  int

#else   /* TURBOC time telling, etc. */
#include <time.h>
#include <stdlib.h>
#define INT_32  long
#define     HZ   CLK_TCK
#endif

#define FIRST_PASS_FLAG FALSE   /*
        whether input gets echoed */

#if UNIX
#define HEAPSIZE   100000          /* #
        nodes in heap */
#define SPINESIZE   100000          /* # cells
        in spine stack */
#define UNIX_REGISTER  register
#else
#define HEAPSIZE   3500          /* # nodes
        in heap */
#define SPINESIZE   500          /* # cells in
        spine stack */
#define UNIX_REGISTER /* no-op */
#endif

#define  MAX_TOKEN 255
#define PROGSIZE   HEAPSIZE/2       /* max
        size of initial program */

#if VAX_ASM
#define HEAP_STRIDE 3
```

```
#else
#define HEAP_STRIDE 2
#endif

typedef union Valuetype {
    struct Celltype *child ;        /* child in
        combinator tree */
    struct Celltype *ptr ;          /* pointer when
        used in as intr ptr */
    INT_32 comb ;
    INT_32 literal  ;
    }  Valuetype ;

typedef struct Celltype {
    Valuetype value ;
    }  Celltype ;

extern Valuetype Evaluate(Valuetype root,
        Celltype **spine_stack);
extern void exit(int exit_value);

extern void dumpgraph(Celltype *root, int
        indent);
extern void spine_under(char *xx);
extern void spine_overflow();

extern void collect_garbage(Celltype
        **top_spine);
extern void fixup_stack(Celltype **top_spine);
extern void init_heap();  /* init heap memory - -
        leaves first xx nodes free */
extern void zapflags();  /* set all flags in heap
        arrays to false */
extern INT_32 mapval(int inptoken);
extern Celltype *newnode(Celltype
        **top_spine);
extern Celltype *newnode_continue(Celltype
        **top_spine);
extern void newnode2(Celltype **top_spine,
         Celltype **nodea, Celltype **nodeb);
extern void newnode2_continue(Celltype
        **top_spine,
         Celltype **nodea, Celltype **nodeb);
extern void newnode3(Celltype **top_spine,
         Celltype **nodea, Celltype **nodeb,
        Celltype **nodec);
extern void newnode3_continue(Celltype
        **top_spine,
         Celltype **nodea, Celltype **nodeb,
        Celltype **nodec);

extern Celltype do_lit_value,do_i_value ; /*
        save DO_LIT and DO_I values */
extern int error_flag ;
extern Celltype *temp_pointer ;

extern Valuetype ip ;
extern Celltype *root_save ;            /* root of
        evalution tree */
extern Celltype *next_free_node ;       /* heap
        free list pointer */
extern Celltype *heap_start, *heap_end ;  /*
        boundaries of in-use heap */
extern Celltype *from_heap, *to_heap ;    /*
        two heap buffers */
extern Celltype **spine_start,**spine_end ;  /*
        boundaries of spine memory */
extern Celltype **spine_stack ; /* points to
        topmost element on spine stack */
extern Celltype
        *spine_mem[SPINESIZE+100];    /*
        spine stack storage */
extern Celltype **fixup_start ;

extern long count;   /* counts number of
        combinators used */
extern int gc ;      /* counts number of garbage
        collections used */
extern int error_flag ;  /* flag if graph has an
        error */

extern long i_counts[MAX_TOKEN+10]; /*
        combinator freq. counts */
extern long i_heap   ;  /* number of heap nodes
        allocated */

/* Use this macro before first reference to me
        node */
#define  Use_Me      me_ptr = *temp_spine ;
/* Rme points to right-hand side of current
        node */
#define  Rme      (me_ptr)->value
#define  Lme      (me_ptr-1)->value

/* Use this macro before first reference to
        parent node */
#define  Use_Parent  parent_ptr =
        *(temp_spine + 1) ;
/* Rparent points to right-hand side of parent
        node */
#define  Rparent     (parent_ptr)->value
#define  Lparent     (parent_ptr-1)->value

/* Use this macro before first reference to
        grandparent node */
#define  Use_Grandparent  /* nop */
/* Rparent points to right-hand side of
        grandparent node */
#define  Rgrandparent     (*(temp_spine +
        2))->value
#define  Lgrandparent     ((*(temp_spine +
        2))-1)->value

/* Use this macro before first reference to
        greatgrandparent node */
#define  Use_Greatgrandparent  /* nop */
/* Rparent points to right-hand side of
        greatgrandparent node */
#define  Rgreatgrandparent     (*(temp_spine +
        3))->value
```

```
#define Lgreatgrandparent    ((*(temp_spine
        + 3))-1)->value

/* Rtemp1 points to right-hand side of 1st temp
        node */
#define Ltemp1    temp1->value
#define Rtemp1    (temp1+1)->value
/* Rtemp2 points to right-hand side of 2nd
        temp node */
#define Ltemp2    temp2->value
#define Rtemp2    (temp2+1)->value
/* Rtemp3 points to right-hand side of 3rd
        temp node */
#define Ltemp3    temp3->value
#define Rtemp3    (temp3+1)->value

#define Pop_Spine(x)  temp_spine += x

/* token definitions for combinators */
#define DO_I     1
#define DO_K     2
#define DO_S     3
#define DO_SPRIME 4
#define DO_B     5
#define DO_C     6
#define DO_IF    7
#define DO_LIT   8
#define DO_PLUS  9
#define DO_MINUS 10
#define DO_LESS  11
#define DO_TIMES 12
#define DO_ONE   13
#define DO_ZERO  14
#define DO_EQ    15
#define DO_GREAT 16
#define DO_P     17
#define DO_U     18
#define DO_NOT   20
#define DO_ABS   21
#define DO_TRUE  22
#define DO_FALSE 23
#define DO_NIL   24
#define DO_PRINT 25
#define DO_NL    26
#define DO_AND   27
#define DO_OR    28
#define DO_MOD   29
#define DO_BSTAR 30
#define DO_CPRIME 31
#define DO_TWO   32
#define DO_THREE 33
#define DO_DIV   34
#define SPINE_UNDERFLOW 35   /* spine
        stack pointer for stack underflow */

#define DO_S1    36
#define DO_S2    37
#define DO_S3    38
#define DO_S4    39
#define DO_S5    40
#define DO_S6    41
#define DO_S7    42
#define DO_S8    43
#define DO_FLOOR    44

#define NOT_A_TOKEN 1001
#define CYCLE       1002
#define LPAREN      1003
#define RPAREN      1004
#define EOF_TOKEN   1005
#define KEYWORD     1006
#define DO_NODE     1007
#define DO_UNWIND   0


#if VAX_ASM
/*   jsb occupies 2 bytes before the lhs when
        fudged*/
#define FUDGE(x)       ( (Celltype *) (
        ((INT_32) (x-1)) |  2))
#define UNFUDGE(x)     ( (Celltype *) (
        ((INT_32) (x+1)) & -3))
/* forwarded cell has bit 1 cleared */
#define MAKE_FORWARD(x) ( (Celltype *) (
        ((INT_32) (x  )) & -3))
#define UNFORWARD(x)    ( (Celltype *) (
        ((INT_32) (x  )) |  2))

#define IS_PTR(x)       ( ! ( ((INT_32) (x)) & 1) )
#define IS_COMB(x)         ( ((INT_32) (x)) & 1)
#define IS_FORWARDED(x)  ( ! ( ((INT_32)
        (x)) & 3) )

#else

#define FUDGE(x)     x
#define UNFUDGE(x)   x
#define MAKE_FORWARD(x) ( (Celltype *) (
        ((INT_32) (x)) |  2))
#define UNFORWARD(x)    ( (Celltype *) (
        ((INT_32) (x)) & -3))
#define IS_PTR(x)       ( ! ( ((INT_32) (x)) &
        0x80000001) )
#define IS_COMB(x)         ( ((INT_32) (x)) &
        0x80000001)
#define IS_FORWARDED(x)     ( ((INT_32)
        (x)) & 2)

#endif


#if UNIX
/* UNMARKED corresponds to a subroutine
        call instruction jsb immediate long */
#define UNMARKED 0x9F160101
/* note: 01 is a VAX no-op */
#else
/* need a 16-bit quantity for the PC */
#define UNMARKED    1234
#endif
```

```
#define MARKED     1
```

B.2. KERNEL.C

```
/* TIGRE implementation:  KERNEL.C */
/* kernel for reduction engine */

/* (C) Copyright 1989   Philip Koopman Jr.  */
/*  Last update:  4/17/89   */

/* For historical reasons, this code does not use
 * temp0.  Therefore, temporary pointers to
          nodes are contained
 * in the variables temp1, temp2, temp3, ...
 */

#include "reduce.h"
#include "heap.h"
#include "sim.h"

Celltype *nodea,*nodeb,*nodec;

Valuetype breakval;

/* ____________________________________ */
Valuetype Evaluate(root,spine_ptr)
  Valuetype root; Celltype **spine_ptr;
 {
   UNIX_REGISTER Valuetype result,resultb;
   UNIX_REGISTER Valuetype ip;
   register Celltype **temp_spine;
   UNIX_REGISTER Celltype *parent_ptr,
         *me_ptr;
   UNIX_REGISTER Celltype *temp1,*temp2;
   UNIX_REGISTER Celltype *temp3;

   ip = root;
   temp_spine = spine_ptr;

   do
    {  /* important:  code and data must be in
          same address space!! */

      while ( ! (ip.comb & 1) )  /*  cures an
        R2000 cc bug */
/*      while ( IS_PTR(ip.ptr) ) */
       {
         ip.ptr = UNFUDGE(ip.ptr);
         *(- -temp_spine) = ip.ptr+1;   /* push
        node onto spine stack */
         ip = ip.ptr->value;   /* set ip to child of
        left node */
       }

    /* now we point to a combinator node */
    count += 1;

    switch(ip.comb >> 2)  /* convert back to
          normal integer */
     {

/* ____________________________________ */

case DO_I:
Use_Me;
ip = Rme;
Pop_Spine(1);
continue;

/* ____________________________________ */

case DO_K:
Use_Me;
ip  = Rme;
Pop_Spine(2);
continue;

/* ____________________________________ */

case DO_S:
NEWNODE2;
/* non-garbage collection code */
Use_Me;
Use_Parent;
Ltemp1 = ip     = Rme;
Ltemp2          = Rparent;
Pop_Spine(1);
Use_Parent;
Rtemp1 = Rtemp2 = Rparent;
Rparent.child = FUDGE(temp2);
Lparent.child = FUDGE(temp1);
*(temp_spine) = temp1+1;
continue;

/* ____________________________________ */

case DO_SPRIME:
NEWNODE3;
Use_Me;
Use_Parent;
Ltemp1  = ip            = Rme;
Rtemp1.child  = FUDGE(temp2);
Ltemp2        = Rparent;
Pop_Spine(2);
Use_Me;
Use_Parent;
Ltemp3 = Rme;
Lparent.child = FUDGE(temp1);
Rtemp2  = Rtemp3  = Rparent;
Rparent.child = FUDGE(temp3);
*(temp_spine) = temp1+1;
continue;
```

```
/* ____________________________ */

case DO_B:
NEWNODE1;
Use_Me ;
Use_Parent;
ip          = Rme ;
Ltemp1      = Rparent  ;
Pop_Spine(2) ;
Use_Me ;
Lme = ip ;
Rtemp1      = Rme ;
Rme.child = FUDGE(temp1) ;
continue;

/* ____________________________ */

case DO_C:
NEWNODE1;
Use_Me ;
Ltemp1      = ip            = Rme  ;
Pop_Spine(1);
Use_Parent;
Use_Me ;
Rtemp1      = Rparent ;
Rparent     = Rme ;
Lparent.child  = FUDGE(temp1) ;
*(temp_spine) = temp1+1 ;
continue ;

/* ____________________________ */

case DO_IF:
Use_Me ;
result = Evaluate(Rme,temp_spine);
Pop_Spine(1);
Use_Me ;
Use_Parent;
if (result.literal)
 {
   ip = Rparent = Rme ;
   Lparent.comb = do_i_value.value.comb ;
   Pop_Spine(2);
   continue ;
 }
ip = Rparent ;
Lparent.comb = do_i_value.value.comb ;
Pop_Spine(2);
continue ;

/* ____________________________ */

case DO_LIT:
Use_Me ;
result = Rme ;
Pop_Spine(1) ;
break ;

/* ____________________________ */

case DO_ZERO:
result.literal = 0;
break ;

/* ____________________________ */

case DO_NIL:
result.literal = -131313 ;
break ;

/* ____________________________ */

case DO_NOT:
Use_Me ;
result  =  Evaluate(Rme,temp_spine) ;
Use_Me ;
result.literal = ! result.literal ;
Rme = result ;
Lme.comb    = do_lit_value.value.comb ;
Pop_Spine(1) ;
break ;

/* ____________________________ */

case DO_PLUS:
Use_Me ;
result = Evaluate(Rme,temp_spine) ;
Use_Parent;
resultb = Evaluate(Rparent,temp_spine);
Use_Parent;
result.literal = result.literal + resultb.literal ;
Rparent = result ;
Lparent.comb  = do_lit_value.value.comb ;
Pop_Spine(2) ;
break ;

/* ____________________________ */

case DO_AND:
Use_Me ;
result = Evaluate(Rme,temp_spine) ;
if(result.literal)
 { Use_Parent;
   resultb = Evaluate(Rparent,temp_spine);
   if (resultb.literal) result.literal = TRUE ;
   else    result.literal = FALSE ;
 }
 else    result.literal = FALSE ;
Use_Parent;
Rparent = result ;
Lparent.comb = do_lit_value.value.comb ;
Pop_Spine(2) ;
break ;

/* ____________________________ */

case DO_OR:
Use_Me ;
result = Evaluate(Rme,temp_spine) ;
```

```
if(!result.literal)
 { Use_Parent;
   resultb = Evaluate(Rparent,temp_spine);
   if (!resultb.literal) result.literal = FALSE ;
   else    result.literal = TRUE ;
 }
 else    result.literal = TRUE ;
Use_Parent;
Rparent = result ;
Lparent.comb = do_lit_value.value.comb ;
Pop_Spine(2) ;
break ;


/* _____________________________ */

case DO_P:
/*  P returns the address of the parent node
       RHS as its
 *  result.  Also, all I-nodes between the
       parent and
 *  me nodes are shorted by re-writing
       Lparent.
 * 1) comparisons can compare on this
       address.
 * 2) The U operator can then look at the
       result & LHS
 */
Use_Parent;
Use_Me ;
result.child = parent_ptr - 1 ;
Lparent.child = FUDGE(me_ptr - 1);
Pop_Spine(2);
break ;

/* _____________________________ */

case DO_U:
/* Evaluate Rparent subtree.  That subtree's
       P combinator
 * will return pointer to RHS of node, and
       guarantee that
 * the LHS of that node points to the node
       whose RHS
 * contains the other pair parameter.
 */
Use_Parent;
/* use temp2 as a scratch pointer to the P
       subtree */
/* apparent bug in TURBOC won't allow
       temp2 = Evaluate(...).ptr ; */
result = Evaluate(Rparent,temp_spine) ;
NEWNODE1;

temp2 = result.ptr ;
Use_Parent ;
Use_Me ;
Lparent.child = FUDGE(temp1) ;
Rparent = Rtemp2 ;
Ltemp1 = ip     = Rme ;
 Rtemp1        =
       (UNFUDGE(Ltemp2.child)+1)->value;
 *(temp_spine) = temp1+1 ;
 continue ;

 /* _____________________________ */

/* Supercombinator definition for fib */

#define fast_mapval(x)  ((x * 4) + 1)

case DO_S3:
 Use_Me;
 result = Evaluate(Rme,temp_spine) ;
 if (result.literal < 3 )
  { result.literal = 1 ;
    Pop_Spine(1);
    break ;
  }

 NEWNODEN(6);
 Use_Me;

 resultb = Rme ;

 (temp1+0)->value.comb =
        fast_mapval(DO_LIT);
 (temp1+1)->value.literal = result.literal - 1 ;

 (temp1+2)->value.comb =
        fast_mapval(DO_S3);
 (temp1+3)->value.child = FUDGE(temp1+0);

 (temp1+4)->value.comb =
        fast_mapval(DO_PLUS);
 (temp1+5)->value.child = FUDGE(temp1+2);

 (temp1+6)->value.comb =
        fast_mapval(DO_LIT);
 (temp1+7)->value.literal = result.literal - 2 ;

 (temp1+8)->value.comb =
        fast_mapval(DO_S3);
 (temp1+9)->value.child = FUDGE(temp1+6);

 (temp1+10)->value.child = FUDGE(temp1+4);
 (temp1+11)->value.child = FUDGE(temp1+8);

 Lme.comb = ip.comb = fast_mapval(DO_I);
 Rme.child = FUDGE(temp1+10);
 continue;

 /* _____________________________ */
 /* Supercombinator definition for nfib */

case DO_S6:
 Use_Me;
 result = Evaluate(Rme,temp_spine) ;
 if (result.literal < 2 )
  { result.literal = 1 ;
```

```
   Pop_Spine(1);
   break ;
 }
 NEWNODEN(8);

 Use_Me;
 resultb = Rme ;
 (temp1+0)->value.comb =
        fast_mapval(DO_LIT);
 (temp1+1)->value.literal = result.literal - 1 ;

 (temp1+2)->value.comb =
        fast_mapval(DO_S6);
 (temp1+3)->value.child = FUDGE(temp1+0);

 (temp1+4)->value.comb =
        fast_mapval(DO_PLUS);
 (temp1+5)->value.child = FUDGE(temp1+2);

 (temp1+6)->value.comb =
        fast_mapval(DO_LIT);
 (temp1+7)->value.literal = result.literal - 2 ;

 (temp1+8)->value.comb =
        fast_mapval(DO_S6);
 (temp1+9)->value.child = FUDGE(temp1+6);

 (temp1+10)->value.child = FUDGE(temp1+4);
 (temp1+11)->value.child = FUDGE(temp1+8);

 (temp1+12)->value.comb =
        fast_mapval(DO_PLUS);
 (temp1+13)->value.comb =
        fast_mapval(DO_ONE);

 (temp1+14)->value.child =
        FUDGE(temp1+12);
 (temp1+15)->value.child =
        FUDGE(temp1+10);

 Lme.comb= ip.comb = fast_mapval(DO_I);
 Rme.child = FUDGE(temp1+14);
 continue;

  /* _____________________________ */

 case SPINE_UNDERFLOW: printf("\n\n
        spine stack underflow\n");
   exit(-1);

..... NOTE: other cases elided .....


 default: printf("\n\n jump to bad
        combinator\n");
   exit(-1);
 }
  return(result);
}
  while ( TRUE ) ;
 /* unreachable code   - -  return(result);  */
}

/* ____________________ */

INT_32 mapval(toknum)
/*  this translates token integers to execution
        addresses */
 int toknum ;
 { int temp_toknum ;
   temp_toknum = toknum << 2 ;  /* force to
        odd value */
   temp_toknum += 1 ;
   return(temp_toknum);}
```

B.3. TIGRE.S

```
#NO_APP

 # /* GRAPH REDUCTION TESTING CODE
        — TIGRE */
 # /* (C) Copyright 1989   Philip Koopman Jr.
        */
 # /*  Last update:  4/17/89   */
 # /*  VAX assembler kernel for reduction
        engine */
 # /*  this version uses the embedded jsr
        commands in the mark word */

.set MARKED,-1
.set UNMARKED,0x9F160101

.set HEAPSTRIDE,3

 # Celltype *nodea,*nodeb,*nodec ;
.comm _nodec,4
.comm _nodeb,4
.comm _nodea,4


 #  r0 - scratch
 #  r1 - result
 #  r2 - resultb
 #  r3 - temp1
 #  r4 - temp2
 #  r5 - temp3
 #  r6 -
 #  r7 - parent_ptr
 #  r8 - me_ptr
 #  r9 - ip
 # r10 - count
```

```
# r11 - next_free_node
# r12 - AP
# r13 - FP
# r14 - SP
# r15 - PC

#     register Valuetype result,resultb ;
#     register Valuetype ip;
#     register Celltype **temp_spine,
          **parent_spine ;
#     register Celltype *temp1,*temp2;
#     register Celltype *temp3;

# Valuetype Evaluate(Valuetype root)
.globl _Evaluate
#  {
_Evaluate:
       .word 0xff4

#     ip = root ;
       movl 4(ap),r9
       movl _next_free_node,r11
       movl _count,r10
       movl _nodea,r0
 # load dummy write location into cache

# call kernel subroutine
      incl _spine_stack  # so that GC will call
          stack cleanup word
       movab -8(sp),_fixup_start
  # starting point of stack fixup for GC
       jsb -2(r9)
      decl _spine_stack

       movl r10,_count
       movl r11,_next_free_node
#  move result to return parameter register
       movl r1,r0
# do a procedure return
       ret


L8:    calls $0,_spine_overflow
       ret

#_____________________________________
#  case DO_I:
       .align 2
       .byte 0
DO_I:
       incl r10
#     ip = Rme ;
#     spine_stack -= 1;
       movl *(sp),r9
#  pop_spine_stack(1)
       movab 4(sp),sp
#     continue;
# note:   jmp *(sp)+ doesn't seem to work
          properly on maxwell
       jmp (r9)

#_____________________________________
#     case DO_K:
       .align 2
       .byte 0
DO_K:
       incl r10
#     ip = Rme ;
#     spine_stack -= 2;
       movl *(sp),r9
#  pop_spine_stack(1)
       movab 8(sp),sp
#     continue;
       jmp (r9)

#_____________________________________
#     case DO_S:
# - -> garbage collection entry point
       .align 2
L13:
       movl r10,_count

       pushab _nodeb
       pushab _nodea
       pushab 8(sp)
       calls $3,_newnode2_continue
       movl _nodea,r3
       movl _nodeb,r4
       movl _next_free_node,r11
       jbr L16
#
# — — — > entry point here < — — -
       .align 2
       .byte 0
DO_S:
       incl r10
#      NEWNODE2;
       movl r11,r3
      movab HEAPSTRIDE*4(r11),r4
       movab 2*HEAPSTRIDE*4(r11),r11
      cmpl r3,_heap_end
       jgequ L13

L16:
       movl (r3),r0
# - - Pre-touch heap node to prevent later
          misses
#     Ltemp1  = ip           = Rme ;
#     Ltemp2                 = Rparent ;
       movl *(sp)+,r9
       movl 4(sp),r7    # r7 now contains parent
          pointer
       movl *(sp),(r4)
       movl (r7),r0
#     parent_spine -= 1 ;
#     Rtemp1  = Rtemp2  = Rparent ;
       movl r0,4(r4)
       movl r9,(r3)
       movl r0,4(r3)
#     Lparent.child = temp1 ;
```

```
        movab -2(r3),-4(r7)
                # re-pointed to mark field
#       Rparent.child = temp2 ;
        movab -2(r4),(r7)
                # re-pointed to mark field

#       *(- -temp_spine) = temp1+1 ;
        movab 4(r3),(sp)
#       continue ;
#  write flush not required - - sp update
           flushed the node
#       movl r0,_nodea  # flush write buffer
        jmp (r9)


#_____________________________________
#       case DO_B:
# - -> garbage collection entry point
        .align 2
L24:
        movl r10,_count

        pushab 0(sp)
        calls $1,_collect_garbage
        movl _next_free_node,r3
        movab HEAPSTRIDE*4(r3),r11
        jbr L25
#
# — — — > entry point here < — — -
        .align 2
        .byte 0
DO_B:
        incl r10
#       NEWNODE1;
        movl r11,r3
        movab HEAPSTRIDE*4(r11),r11
        cmpl r3,_heap_end
        jgequ L24
L25:
        movl (r3),r0
# - - Pre-touch heap node to prevent later
           misses
#       ip          = Rme ;
        movl *(sp)+,r9
#       Ltemp1      = Rparent  ;
        movl *(sp)+,(r3)
#       temp_spine -= 2;
#       Lme = ip ;
        movl (sp),r8
        movl r9,-4(r8)
#       Rtemp1      = Rme ;
        movl (r8),4(r3)
#       Rme.child = temp1 ;
        movab -2(r3),(r8) # re-point to mark field
#       continue;
        movl r0,_nodea  # flush write buffer
        jmp (r9)

#_____________________________________
#       case DO_C:
# - -> garbage collection entry point
        .align 2
L31:
        movl r10,_count

        pushab 0(sp)
        calls $1,_collect_garbage
        movl _next_free_node,r3
        movab HEAPSTRIDE*4(r3),r11
        jbr L32
#
# — — — > entry point here < — — -
        .align 2
        .byte 0
DO_C:
        incl r10
#       NEWNODE1
        movl r11,r3
        movab HEAPSTRIDE*4(r11),r11
        cmpl r3,_heap_end
        jgequ L31

L32:
        movl (r3),r0
        # - - Pre-touch heap node to prevent later
           misses
#       Ltemp1      = ip            = Rme  ;
#       temp_spine -= 1;
#       Rtemp1      = Rparent ;
        movl *(sp)+,r9
        movl 4(sp),r7
        movl r9,(r3)
        movl (r7),4(r3)
#       Rparent     = Rme ;
        movl *(sp),(r7)
#       Lparent.child  = temp1 ;
        movab -2(r3),-4(r7)
                # re-point to mark field
#       *(spine_stack) = temp1+1 ;
        movab 4(r3),(sp)
#       continue ;
#  write flush not required - - sp update
           flushed the node
#       movl r0,_nodea  #  flush write buffer
        jmp (r9)

#_____________________________________
#       case DO_IF:
        .align 2
        .byte 0
DO_IF:
        incl r10
#       result = Evaluate(Rme);
        movl *(sp),r9
        jsb (r9)
        movab 4(sp),sp

#       temp_spine -= 1 ;
#        parent_spine = temp_spine - 1 ;
        movl 4(sp),r7
```

```
 #  r7 now has address of parent

 #     if (result.literal)
        tstl r1
        jeql L39
  # if 1st parameter was true, leave node alone
 #       { Rparent = Rme ;
        movl *(sp),(r7)
 #      }
L39:
 #     Lparent.comb = mapval(DO_I) ;
        movl $DO_I,-4(r7)
 #     ip = Rparent ;
        movl (r7),r9
 #     spine_stack -= 3;
        addl2 $8,sp
 #     continue ;
        movl r0,_nodea  #  flush write buffer
        jmp (r9)

#_____________________________
 #     case DO_LIT:
        .align 2
        .byte 0
DO_LIT:
        incl r10
 #     result = Rme ;
 #     spine_stack += 1 ;
        movl *(sp)+,r1
 #     break ;
        rsb

#_____________________________
 #     case DO_FALSE:
 #     case DO_ZERO:
 #     result.literal = 0;
 #     result.literal = FALSE;
 #     break ;
        .align 2
        .byte 0
DO_FALSE:
DO_ZERO:
        incl r10
        clrl r1
        rsb

#_____________________________
 #     case DO_TRUE:
 #     case DO_ONE:
 #     result.literal = TRUE ;
 #     result.literal = 1 ;
        .align 2
        .byte 0
DO_TRUE:
DO_ONE:
        incl r10
        movl $1,r1
 #     break ;
        rsb

#_____________________________
 #     case DO_NOT:
        .align 2
        .byte 0
DO_NOT:
        incl r10
 #     result = Evaluate(Rme) ;
        movl *(sp),r9
        movab -4(sp),sp
        jsb (r9)
        movab 4(sp),sp

 #     result.literal = ! result.literal ;
        clrl r0
        tstl r1
        jneq L47
        incl r0
L47:  movl r0,r1
        movl (sp)+,r7
  #  r7 now has address of me node
 #     spine_stack += 1 ;  (out of order, r7 need
          not be changed)
 #     Lme.comb  = mapval(DO_LIT) ;
        movl $DO_LIT,-4(r7)
 #     Rme = result ;
        movl r1,(r7)
 #     break ;
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________
 #     case DO_PLUS:
        .align 2
        .byte 0
DO_PLUS:
        incl r10
 #     result = Evaluate(Rme) ;
        movl *(sp),r9
 # - - leaves an open stack param
        jsb (r9)

 #     resultb = Evaluate(Rparent);
        movl *4(sp),r9
        movl r1,(sp)
        jsb (r9)

 #     result.literal = result.literal +
          resultb.literal ;
        addl2 (sp),r1
 #      parent_spine = temp_spine - 1 ;
        movl 4(sp),r7         #  r7 now has address of
          parent
        movab 8(sp),sp
 #     spine_stack += 2 ;  (out of order, r7 need
          not be changed)
 #     Lparent.comb  = mapval(DO_LIT) ;
        movl $DO_LIT,-4(r7)
 #     Rparent = result ;
        movl r1,(r7)
 #     break ;
```

```
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________
#     case DO_LESS:
        .align 2
        .byte 0
DO_LESS:
        incl r10
#     result = Evaluate(Rme) ;
# - -  movl *(sp)+,r9
# - -  movab -4(sp),sp
        movl *(sp),r9
        jsb (r9)
        movab 4(sp),sp

#     resultb = Evaluate(Rparent);
        movl *(sp),r9
        pushl r1
        jsb (r9)

#     result.literal = result.literal <
          resultb.literal ;
        clrl r0
        cmpl (sp),r1
        jgeq L57
        incl r0
L57:  movab 8(sp),sp
        movl r0,r1
#       parent_spine = temp_spine - 1 ;
        movl -4(sp),r7
    #  r7 now has address of parent
#     spine_stack += 2 ;  (out of order, r7 need
          not be changed)
#     Lparent.comb  = mapval(DO_LIT) ;
        movl $DO_LIT,-4(r7)
#     Rparent = result ;
        movl r1,(r7)
#     break ;
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________
#     case DO_AND:
        .align 2
        .byte 0
DO_AND:
        incl r10
#     Use_Me ;
#     result = Evaluate(Rme,temp_spine) ;
        movl *(sp),r9
# - - leaves an open stack param
        jsb (r9)
#     if(result.literal)
        tstl r1
        jeql DO_AND1
      # if false, leave false in result register
#       { Use_Parent;
        movl 4(sp),r7
#         resultb =
            Evaluate(Rparent,temp_spine);
        movl (r7),r9
        jsb (r9)
        tstl r1
        jeql DO_AND1
  # if false, leave false in result register
#         if (resultb.literal) result.literal = TRUE ;
#         else    result.literal = FALSE ;
        movl $1,r1
#       }
#       else    result.literal = FALSE ;
DO_AND1:

#     Use_Parent;
        movl 4(sp),r7
#     Rparent = result ;
#     Lparent.comb = do_lit_value.value.comb ;
        movl $DO_LIT,-4(r7)
        movl r1,(r7)
#     Pop_Spine(2) ;
        movab 8(sp),sp
#     break ;
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________
#     case DO_OR:
        .align 2
        .byte 0
DO_OR:
        incl r10
#     Use_Me ;
#     result = Evaluate(Rme,temp_spine) ;
        movl *(sp),r9
# - - leaves an open stack param
        jsb (r9)
#     if(!result.literal)
        tstl r1
        jneq DO_OR1
  # if true, leave true in result register
#       { Use_Parent;
        movl 4(sp),r7
#         resultb =
            Evaluate(Rparent,temp_spine);
        movl (r7),r9
        jsb (r9)
        tstl r1
#         if (!resultb.literal) result.literal =
            FALSE ;
#         else    result.literal = TRUE ;
#       }
#       else    result.literal = TRUE ;
        jeql DO_OR2
  # if false, leave false in result register
DO_OR1:
        movl $1,r1

DO_OR2:
#     Use_Parent;
        movl 4(sp),r7
```

```
#     Rparent = result ;
#     Lparent.comb = do_lit_value.value.comb ;
        movl $DO_LIT,-4(r7)
        movl r1,(r7)
#     Pop_Spine(2) ;
        movab 8(sp),sp
#     break ;
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________________
#     case DO_P:
        .align 2
        .byte 0
DO_P:
        incl r10
#     /*  P returns the address of the parent
            node RHS as its
#     *  result.  Also, all I-nodes between the
            parent and
#     *  me nodes are shorted by re-writing
            Lparent.
#     * 1) comparisons can compare on this
            address.
#     * 2) The U operator can then look at the
            result & LHS
#     */
#     Use_Parent;
        movl 4(sp),r7
#     Use_Me ;
        movl (sp),r8
#     result.child = parent_ptr - 1 ;
        movab -4(r7),r1
#     Lparent.child = FUDGE(me_ptr - 1);
        movab -6(r8),(r1)
#     Pop_Spine(2);
        movab 8(sp),sp
#     break ;
        movl r0,_nodea  #  flush write buffer
        rsb

#_____________________________________
#     case DO_U:
# - -> garbage collection entry point
        .align 2
DO_U_G:
        movl r10,_count

        pushab 0(sp)
        calls $1,_collect_garbage
        movl _next_free_node,r3
        movab HEAPSTRIDE*4(r3),r11
        jbr DO_U_CONT
#
# — — — > entry point here < — — -
        .align 2
        .byte 0
DO_U:
        incl r10
#     /* Evaluate Rparent subtree.  That
            subtree's P combinator
#     * will return pointer to RHS of node, and
            guarantee that
#     * the LHS of that node points to the node
            whose RHS
#     * contains the other pair parameter.
#     */
#     Use_Parent;
        movl 4(sp),r7
#     /* use temp2 as a scratch pointer to the P
            subtree */
#     /* apparent bug in TURBOC won't allow
            temp2 = Evaluate(...).ptr ; */
#     result = Evaluate(Rparent,temp_spine) ;
        movl (r7),r9
        movab -4(sp),sp
 # dummy entry for garbage collection
        jsb (r9)

        movab 4(sp),sp

#     NEWNODE1
        movl r11,r3
       movab HEAPSTRIDE*4(r11),r11
       cmpl r3,_heap_end
        jgequ DO_U_G

DO_U_CONT:
        movl (r3),r0
  # Pre-touch heap node to prevent later misses

#     temp2 = result.ptr ;      # just use r1
#     Use_Parent ;
        movl 4(sp),r7
#     Use_Me ;
        movl (sp),r8
#     Lparent.child = FUDGE(temp1) ;
        movab -2(r3),-4(r7)
#     Rparent = Rtemp2 ;
        movl 4(r1),(r7)
#     Ltemp1 = ip     = Rme ;
        movl (r8),r9
        movl r9,(r3)
#     Rtemp1          = (UNFUDGE(
            Ltemp2.child)+1)->value;
        movl (r1),r0
        movl 6(r0),4(r3)
#     *(temp_spine) = temp1+1 ;
        movab 4(r3),(sp)
#     continue ;
#  write flush not required - - sp update
            flushed the node
#       movl r0,_nodea  # flush write buffer
        jmp (r9)

#/* _____________________________________ */
# nfib supercombinator
#case DO_S6:
# - -> garbage collection entry point
        .align 2
```

```
LS6:
        movl r10,_count
        pushab (sp)
        pushl $8
        calls $2,_newnoden_continue
        movl _next_free_node,r11
        jbr LS6A
#
# — — — > entry point here < — — -
        .align 2
        .byte 0
DO_S6:
        incl r10
#  Use_Me;
#  result = Evaluate(Rme,temp_spine) ;
        movl *(sp),r9
        movab -4(sp),sp
        jsb (r9)
        movab 4(sp),sp
#  if (result.literal < 2 )
        cmpl r1,$2
        bgeq S6_CONT
#   { result.literal = 1 ;
        movab 4(sp),sp
        movl $1,r1
#     Pop_Spine(1);
#     break ;
        rsb
#   }

S6_CONT:
LS6A:
#  NEWNODEN(8);
        movl r11,r3
        movab 8*HEAPSTRIDE*4(r11),r11
       cmpl r11,_heap_end
        jgequ LS6
#       movl (r3),r0
        # - - Pre-touch heap node to prevent later
            misses
#       movl 16(r3),r1
#       movl 32(r3),r0
#       movl 48(r3),r1
#       movl 64(r3),r0
#       movl 80(r3),r1
#
#  Use_Me;
#  resultb = Rme ;
        movl (sp),r8
#       movl (r8),r2
#  (temp1+0)->value.comb =
            fast_mapval(DO_LIT);
        movl $DO_LIT, (r3)
#  (temp1+1)->value.literal = result.literal - 1 ;
        movab -1(r1), 4(r3)
#
#  (temp1+3)->value.comb =
            fast_mapval(DO_S6);
        movl $DO_S6, 12(r3)
#  (temp1+4)->value.child = FUDGE(temp1+0);
        movab -2(r3), 16(r3)
#
#  (temp1+6)->value.comb =
            fast_mapval(DO_PLUS);
        movl $DO_PLUS, 24(r3)
#  (temp1+7)->value.child = FUDGE(temp1+3);
        movab 10(r3), 28(r3)
#
#  (temp1+9)->value.comb =
            fast_mapval(DO_LIT);
        movl $DO_LIT, 36(r3)
#  (temp1+10)->value.literal = result.literal - 2
            ;
        movab -2(r1), 40(r3)
#
#  (temp1+12)->value.comb =
            fast_mapval(DO_S6);
        movl $DO_S6, 48(r3)
#  (temp1+13)->value.child =
            FUDGE(temp1+9);
        movab 34(r3), 52(r3)
#
#  (temp1+15)->value.child =
            FUDGE(temp1+6);
        movab 22(r3), 60(r3)
#  (temp1+16)->value.child =
            FUDGE(temp1+12);
        movab 46(r3), 64(r3)
#
#  (temp1+18)->value.comb =
            fast_mapval(DO_PLUS);
        movl $DO_PLUS, 72(r3)
#  (temp1+19)->value.comb =
            fast_mapval(DO_ONE);
        movl $DO_ONE, 76(r3)
#
#  (temp1+21)->value.child =
            FUDGE(temp1+18);
        movab 70(r3), 84(r3)
#  (temp1+22)->value.child =
            FUDGE(temp1+15);
        movab 58(r3), 88(r3)
#
#  Lme.comb= ip.comb = fast_mapval(DO_I);
        movl $DO_I, r9
        movl $DO_I, -4(r8)
#  Rme.child = FUDGE(temp1+21);
        movab 82(r3), (r8)
#  continue;
        jmp (r9)
#/* _____________________________ */
#
#
..... NOTE: other cases elided .....

#_____________________________
#
#    case SPINEUNDERFLOW: printf("\n\n
            spine stack underflow\n");
#       exit(-1);
```

```
        .align 2
        .byte 0
SPINE_UNDER:
        pushab LC2
        calls $1,_printf
        mnegl $1,-(sp)
        calls $1,_exit
        jbr TEMPRSB
#_____________________________
#    default: printf("\n\n jump to bad
           combinator\n");
#       exit(-1);
#    }
        .align 2
        .byte 0
BAD_COMBINATOR:
        pushab LC3
        calls $1,_printf
        mnegl $1,-(sp)
        calls $1,_exit
        jbr TEMPRSB
#_____________________________
#     return(result);
#   }
#     while ( TRUE ) ;
#     return(result);
#   }
L9:

 # ensure that temp_spine is maintained
TEMPRSB:
        movl _next_free_node,r11
        movl _spine_stack,r8
        rsb

#_____________________________
# int mapval(int toknum )
#   {  return(toknum);}
# /*  this translates token integers to
           execution addresses */
        .align 1
.globl _mapval
_mapval:
        .word 0x0
        movl 4(ap),r0
        addl2 r0,r0
        addl2 r0,r0
        jcc Lmap2
        movl $END_TABLE-TABLE,r0
Lmap2:
        movab TABLE(r0),r0
        cmpl r0,$END_TABLE
        jlss Lmap1
        movl $END_TABLE,r0
Lmap1:
        movl (r0),r0
        ret

 #   NOTE:  all entry points are constrained to
           be aligned as
 #          evenly divisible by 4 + 1
        .align 2
        .long BAD_COMBINATOR  #
        .long BAD_COMBINATOR  #
TABLE:
        .long BAD_COMBINATOR   #0th element
        .long DO_I  # 1
        .long DO_K  # 2
        .long DO_S  # 3
        .long DO_SPRIME  # 4
        .long DO_B  # 5
        .long DO_C  # 6
..... NOTE: other cases elided .....
        .long BAD_COMBINATOR  #
END_TABLE:
        .long BAD_COMBINATOR  #
1
```

B.4. MIPS.S

```
        .verstamp  1 31


# TIGRE implementation:  MIPS.S
# MIPS R2000 version of kernel for reduction
           engine


# (C) Copyright 1989   Philip Koopman Jr.
#  Last update:  4/17/89


# REGISTER ASSIGMENTS:  (* indicates
           automatically saved register)
#    0 - zero
#    1 - reserved for assembler use (macro
           instructions)
#    2 - result return register
#    3 - xx
#    4 -
#    5 -
#    6 -
#    7 -
#    8 - store scratch
#    9 - scratch
#   10 - ip
#   11 - resulta
#   12 - resultb
#   13 -
#   14 -
#   15 -
# * 16 - temp_spine (spine stack pointer)
```

```
# * 17 - parent_ptr
# * 18 - temp1
# * 19 - temp2
# * 20 - temp3
# * 21 - me_ptr
# * 22 - next_free_node
# * 23 - count
#   24 -
#   25 - load scratch
#   26 - reserved for OS
#   27 - reserved for OS
#   28 - gp, reserved for OS
#   29 - sp
# * 30 - xx
#   31 - ra, link register for subroutine calls
### 76(sp) - ip
### 84(sp) - result

        .globl Evaluate
        .loc    2 17
#  17  {
        .ent    Evaluate 2

#_____________________________
Evaluate:
        .option       O1
        subu $sp, 96
        sd      $4, 96($sp)
        sw      $6, 104($sp)
        sw      $31, 52($sp)
        sd      $22, 44($sp)
        sd      $20, 36($sp)
        sd      $18, 28($sp)
        sd      $16, 20($sp)
        .mask         0x80FF0000, -52
        .frame        $sp, 96, $31
        .loc    2 28
#  28     ip = root ;
        addu $14, $sp, 84
        addu $15, $sp, 100
        .set    noat
        lw      $1, 0($15)
        sw      $1, 0($14)
        .set    at
        .loc    2 29
#  29     temp_spine = spine_ptr ;
        lw      $16, 104($sp)
        lw      $10, 84($sp)   # get ip into r10
        lw      $23, count
        lw      $22, next_free_node
        jal     $MY_EVAL

#_____________________________
#  break - - return to calling routine
$BREAK:
        .loc    2 575
# 575           return(result);
        sw      $11, 92($sp)
        sw      $23, count
        sw      $22, next_free_node
        addu $2, $sp, 92
        lw      $8, 96($sp)
        .set    noat
        lw      $1, 0($2)
        sw      $1, 0($8)
        .set    at
        move $2, $8
        ld      $16, 20($sp)
        ld      $18, 28($sp)
        ld      $20, 36($sp)
        ld      $22, 44($sp)
        lw      $31, 52($sp)
        addu $sp, 96
        j       $31

# ___________________________________
# Lightweight subroutine call routines

$MY_BREAK:
        lw      $31, 0($16)
        addu $16, $16, 4 # pop address
        j       $31

        .set    noreorder
$MY_EVAL:
        addu $16, $16, -4
        sw      $31, 0($16)

#_____________________________
#  Threading loop
$THREAD:
#  30     do
#  31   { /* important: code and data must
                be in same address space!! */
#  32         while ( ! (ip.comb & 1) )
                /* cures an R2000 cc bug */
        bltz    $10, $FOUND_COMB
        addu $9, $10, 4     # delay slot
$THREAD_LOOP:
#  33/*         while ( IS_PTR(ip.ptr) ) */
#  34           {
#  38             ip.ptr = UNFUDGE(ip.ptr) ;
#  39             *(- -temp_spine) = ip.ptr+1 ;
               /* push node onto stack */
#  40             ip = ip.ptr->value ;
               /* set ip to child of left node */
#  44           }
        addu $16, $16, -4
        lw      $10, 0($10)
        sw      $9, 0($16)
        bgez    $10, $THREAD_LOOP
        addu $9, $10, 4   # delay slot
$FOUND_COMB:
#  46 /* now we point to a combinator
              node */
#  47      if (temp_spine < spine_start)
                spine_overflow() ;
#             lw      $15, spine_start
#             bgeu    $16, $15, $35
#             jal     spine_overflow
```

```
#     $35:
# 53      count += 1 ;
# 62      switch(ip.comb)
    # highest bit
      sll     $10, $10, 1
# extra shift right required for now
      srl     $10, $10, 1
      j       $10
      addu $23, $23, 1
      .set    reorder

#_____________________________
# case I
      .loc    2 71
      .set    noreorder
      .align        3
$DO_I:
#       Use_Me ;
      lw      $21,0($16)
# 72    ip = Rme ;
# 73    Pop_Spine(1);
      addu $16, $16, 4
      lw      $10,0($21)
# 74    continue;
      b       $THREAD
      nop       # delay slot - - can't load $10
      .set    reorder
#_____________________________
# case K
      .loc    2 82
      .set    noreorder
      .align        3
$DO_K:
#       Use_Me ;
      lw      $21,0($16)
# 72    ip = Rme ;
# 73    Pop_Spine(2);
      addu $16, $16, 8
      lw      $10,0($21)
# 74    continue;
      b       $THREAD
      nop       # delay slot - - can't load $10
#       Use_Me ;
#     lw      $21,0($16)
# 100           Useparent;
#     lw      $17, 4($16)
# 82    Rparent = ip = Rme ;
# 83    Pop_Spine(2);
#     lw      $10, 0($21)
#     addu $16, $16, 8
#     sw      $10, 0($17)
# 244           Lparent.comb = do_i_value.value.comb
        ;
# 84    continue;
#     lw      $9, do_i_value
#     b       $THREAD
#     sw      $9, -4($17)       # delay slot
      .set    reorder

#_____________________________
# case S
      .loc    2 92
      .set    noreorder
      .align        3
$DO_S:
# 92    NEWNODE2 ;
      lw      $15, heap_end
      addu $18, $22, 0
      addu $19, $22, 8
      bltu    $18, $15, $39
      addu $22, $22, 16     # delay slot
      .set    reorder
# - - garbage collection - -
      sw      $22, next_free_node
      sw      $23, count
      move $4, $16
      la      $5, nodea
      la      $6, nodeb
      jal     newnode2_continue
      lw      $18, nodea
      lw      $19, nodeb
      lw      $22, next_free_node
# — — — — — — — — -
$39:
      .set    noreorder
#       Use_Me ;
      lw      $21, 0($16)
# 100           Useparent;
      lw      $17, 4($16)
# 99    Ltemp1 = ip     = Rme ;
# 101           Ltemp2        = Rparent ;
      lw      $10, 0($21)
      lw      $9, 0($17)
      sw      $10, 0($18)
# 102           Pop_Spine(1);
# 103           Useparent;
# 104           Rtemp1 = Rtemp2 = Rparent ;
      lw      $17, 8($16)
      sw      $9, 0($19)
      lw      $8, 0($17)
      addu $16, $16, 4
      sw      $8, 4($19)
      sw      $8, 4($18)
# 105           Rparent.child = FUDGE(temp2) ;
      sw      $19, 0($17)
# 106           Lparent.child = FUDGE(temp1) ;
      sw      $18, -4($17)
# 107           *(temp_spine) = temp1+1 ;
# 108           continue ;
      addu $9, $18, 4
      b       $THREAD
      sw      $9, 0($16)      # delay slot
      .set    reorder


#_____________________________
# case B
      .loc    2 142
      .align        3
      .set    noreorder
```

```
$DO_B:
 #  NEWNODE1;
       lw     $15, heap_end
       addu $18, $22, 0
       bltu   $18, $15, $B_OK
       addu $22, $22, 8          # delay slot
       .set   reorder
 # - - garbage collection - -
       sw     $22, next_free_node
       sw     $23, count
       move $4, $16
       jal    newnode_continue
       move $18, $2
       lw     $22, next_free_node
 # - — — — — — — — — — — — -
       .set   noreorder
$B_OK:
 #  Use_Me ;
 #  Use_Parent;
       lw     $17, 4($16)
       lw     $21, 0($16)
 #  ip          = Rme ;
 #  Ltemp1      = Rparent  ;
       lw     $9, 0($17)
       lw     $10, 0($21)
       sw     $9, 0($18)
 #  Pop_Spine(2) ;
 #  Use_Me ;
 #  Lme = ip ;
 #  Rtemp1      = Rme ;
       lw     $21, 8($16)
       addu $16, $16, 8
       lw     $8, 0($21)
       sw     $10, -4($21)
       sw     $8, 4($18)
 #  Rme.child = FUDGE(temp1) ;
 #  continue;
       b      $THREAD
       sw     $18, 0($21)      # delay slot
       .set   reorder
 #_____________________
 #  case C
       .loc   2 190
       .align       3
       .set   noreorder
$DO_C:
 #  NEWNODE1;
       lw     $15, heap_end
       addu $18, $22, 0
       bltu   $18, $15, $C_OK
       addu $22, $22, 8          # delay slot
       .set   reorder
 # - - garbage collection - -
       sw     $22, next_free_node
       sw     $23, count
       move $4, $16
       jal    newnode_continue
       move $18, $2
       lw     $22, next_free_node
 # - — — — — — — — — — — — -
       .set   noreorder
$C_OK:
 #  Use_Me ;
 #  Ltemp1      = ip          = Rme  ;
 #  Pop_Spine(1);
 #  Use_Parent;
       lw     $21, 0($16)
       addu $16, $16, 4
       lw     $10, 0($21)
       lw     $17, 4($16)
       sw     $10, 0($18)
 #  Use_Me ;
 #  Rtemp1      = Rparent ;
       lw     $8, 0($17)
       lw     $21, 0($16)
       sw     $8, 4($18)
 #  Rparent     = Rme ;
 #  Lparent.child  = FUDGE(temp1) ;
       lw     $9, 0($21)
       sw     $18, -4($17)
       sw     $9, 0($17)
 #  *(temp_spine) = temp1+1 ;
       addu $8, $18, 4
 #  continue ;
       b      $THREAD
       sw     $8, 0($16)      # delay slot
       .set   reorder

 #_____________________
 #  case IF
       .loc   2 237
       .set   noreorder
       .align       3
$DO_IF:
 #       Use_Me ;
       lw     $21, 0($16)
 #        result = Evaluate(Rme,temp_spine) ;
 #  - - The 'me' slot on the spine stack holds the
         data save item slot.
       jal    $MY_EVAL
       lw     $10, 0($21)      # delay slot

 # 238          Pop_Spine(1);
 #       Use_Me ;
 # 239          Use_Parent;
 # 240          if (result.literal)
       lw     $21, 4($16)
       lw     $17, 8($16)
       beq    $11, $0, $51
       addu $16, $16, 4      # delay slot

 # 241          { ip = Rparent = Rme ;
       lw     $10, 0($21)
       lw     $9, do_i_value
       sw     $10, 0($17)
       addu $16, $16, 8
       b      $THREAD
       sw     $9, -4($17)      # delay slot
 # 242          }
$51:
```

```
# 243           else { ip = Rparent ; }
      lw     $10, 0($17)
# 244           Lparent.comb = do_i_value.value.comb
        ;
# 245           Pop_Spine(2);
# 246           continue ;
      lw     $9, do_i_value
      addu $16, $16, 8
      b      $THREAD
      sw     $9, -4($17)      # delay slot
      .set   reorder

#_____________________________
#  case LIT
      .loc   2 254
      .set   noreorder
      .align       3
$DO_LIT:
#         Use_Me ;
# 254           result = Rme ;
# 255           Pop_Spine(1) ;
# 256           break ;
      lw     $21, 0($16)
      lw     $31, 4($16)
      lw     $11, 0($21)
      j      $31
      addu $16, $16, 8
             # pop address  - - delay slot
      .set   reorder

#_____________________________
#  case NOT
      .loc   2 310
      .align       3
      .set   noreorder
$DO_NOT:
#  Use_Me ;
      lw     $21, 0($16)
#  result  =  Evaluate(Rme,temp_spine) ;
      addi  $16, -4
             # spine stack parameter element
      jal    $MY_EVAL
      lw     $10, 0($21)      # delay slot

      addi  $16,  4
#  Use_Me ;
      lw     $21, 0($16)
#  result.literal = ! result.literal ;
      seq    $11, $11, 0
#  Rme = result ;
      sw     $11, 0($21)
#  Lme.comb    = do_lit_value.value.comb ;
#  Pop_Spine(1) ;
#  break ;
      lw     $9, do_lit_value
      addi  $16, 4
      lw     $31, 0($16)
      sw     $9, -4($21)
      j      $31
      addu $16, $16, 4
                    # pop address  - - delay slot
      .set   reorder

#_____________________________
#  case 1
#  case TRUE
      .loc   2 387
      .set   noreorder
      .align       3
$DO_TRUE:
$DO_ONE:
# 387           result.literal = 1 ;
# 388           break ;
      lw     $31, 0($16)
      addu $11, $0, 1
      j      $31
      addu $16, $16, 4
             # pop address - - delay slot
      .set   reorder

#_____________________________
#  case +
      .loc   2 355
      .set   noreorder
      .align       3
$DO_PLUS:
#         Use_Me ;
      lw     $21, 0($16)
# 355         result = Evaluate(Rme,temp_spine) ;
#  - - The 'me' slot on the spine stack holds the
          data save item.
      jal    $MY_EVAL
      lw     $10, 0($21)      # delay slot

# 356           Useparent;
      lw     $17, 4($16)
# 357           resultb =
         Evaluate(Rparent,temp_spine);
      sw     $11, 0($16)
      jal    $MY_EVAL
      lw     $10, 0($17)      # delay slot

# 358           result.literal = result.literal +
        resultb.literal ;
# 359           Use_Parent;
      lw     $12, 0($16)
      lw     $17, 4($16)
      addu $11, $12, $11
# 360           Rparent = result ;
# 361           Lparent.comb  =
         do_lit_value.value.comb ;
# 362           Pop_Spine(2) ;
# 363           break ;
      lw     $9, do_lit_value
      sw     $11, 0($17)
      lw     $31, 8($16)
      sw     $9, -4($17)
      j      $31
      addu $16, $16, 12
             # pop address & spine stuff
```

```
        .set    reorder

#_____________________________
 #  case AND
        .loc    2 494
        .align          3
        .set    noreorder
$DO_AND:
 #  Use_Me ;
        lw      $21, 0($16)
 #  result = Evaluate(Rme,temp_spine) ;
 #  - - The 'me' slot on the spine stack holds the
           data save item.
        jal     $MY_EVAL
        lw      $10, 0($21)       # delay slot

 #  if(result.literal)
        beq     $11, $0, $AND_FALSE
 #    { Use_Parent;
        lw      $17, 4($16)       #delay slot
 #      resultb = Evaluate(Rparent,temp_spine);
        jal     $MY_EVAL
        lw      $10, 0($17)       # delay slot

 #      if (resultb.literal) result.literal = TRUE ;
 #      else    result.literal = FALSE ;
        beq     $11, $0, $AND_FALSE
        nop       # delay slot
        li      $11, 1
 #    }
 #    else    result.literal = FALSE ;
$AND_FALSE:
 #  Use_Parent;
        lw      $17, 4($16)
 #  Rparent = result ;
 #  Lparent.comb = do_lit_value.value.comb ;
 #  Pop_Spine(2) ;
 #  break ;
        lw      $9, do_lit_value
        sw      $11, 0($17)
        lw      $31, 8($16)
        sw      $9, -4($17)
        j       $31
                  # pop address & spine stuff
        addu $16, $16, 12
        .set    reorder

#_____________________________
 #  case OR
        .loc    2 514
        .align          3
        .set    noreorder
$DO_OR:
 #  Use_Me ;
        lw      $21, 0($16)
 #  result = Evaluate(Rme,temp_spine) ;
 #  - - The 'me' slot on the spine stack holds the
           data save item.
        jal     $MY_EVAL
        lw      $10, 0($21)       # delay slot

 #  if(!result.literal)
        bne     $11, $0, $OR_TRUE
 #    { Use_Parent;
        lw      $17, 4($16)       # delay slot
 #      resultb = Evaluate(Rparent,temp_spine);
        jal     $MY_EVAL
        lw      $10, 0($17)       # delay slot

 #      if (!resultb.literal) result.literal = FALSE ;
 #      else    result.literal = TRUE ;
        beq     $11, $0, $OR_FALSE
        nop       # delay slot
 #    }
 #    else    result.literal = TRUE ;
$OR_TRUE:
        li      $11, 1
$OR_FALSE:
 #  Use_Parent;
        lw      $17, 4($16)
 #  Rparent = result ;
 #  Lparent.comb = do_lit_value.value.comb ;
 #  Pop_Spine(2) ;
 #  break ;
        lw      $9, do_lit_value
        sw      $11, 0($17)
        lw      $31, 8($16)
        sw      $9, -4($17)
        j       $31
        addu $16, $16, 12
                  # pop address & spine stuff
        .set    reorder

#_____________________________
 #  case P
        .loc    2 538
        .align          3
        .set    noreorder
$DO_P:
 #  /*  P returns the address of the parent node
           RHS as its
 #    *  result.  Also, all I-nodes between the
           parent and
 #    *  me nodes are shorted by re-writing
           Lparent.
 #    * 1) comparisons can compare on this
           address.
 #    * 2) The U operator can then look at the
           result & LHS
 #    */
 #  Use_Parent;
        lw      $17, 4($16)
 #  Use_Me ;
        lw      $21, 0($16)
 #  result.child = parent_ptr - 1 ;
        addu $11, $17, -4
 #  Lparent.child = FUDGE(me_ptr - 1);
 #  Pop_Spine(2);
 #  break ;
        addu $8, $21, -4
```

```
        lw      $31, 8($16)
        sw      $8, -4($17)
        j       $31
                # pop address & spine stuff
        addu $16, $16, 12
        .set    reorder

#_____________________________
#  case U
        .loc    2 549
        .align          3
        .set    noreorder
$DO_U:
#  /* Evaluate Rparent subtree.  That
          subtree's P combinator
#   * will return pointer to RHS of node, and
          guarantee that
#   * the LHS of that node points to the node
          whose RHS
#   * contains the other pair parameter.
#   */
#  Use_Parent;
        lw      $17, 4($16)
#  /* use temp2 as a scratch pointer to the P
          subtree */
#  /* apparent bug in TURBOC won't allow
          temp2 = Evaluate(...).ptr ; */
#  result = Evaluate(Rparent,temp_spine) ;
                # spine stack parameter element
        addu $16, $16,  -4
        jal     $MY_EVAL
        lw      $10, 0($17)       # delay slot

        addu $16, $16,   4
#  NEWNODE1;
        lw      $15, heap_end
        addu $18, $22, 0
        bltu    $18, $15, $U_OK
        addu $22, $22, 8            # delay slot

        .set    reorder
# - - garbage collection - -
        sw      $22, next_free_node
        sw      $23, count
        move $4, $16
        jal     newnode_continue
        move $18, $2
        lw      $22, next_free_node
# - — — — — — — — — — — — -
        .set    noreorder
$U_OK:
#  temp2 = result.ptr ;   — just use $11 for
          temp2
#  Use_Parent ;
        lw      $17, 4($16)
#  Use_Me ;
        lw      $21, 0($16)
#  Lparent.child = FUDGE(temp1) ;
        sw      $18, -4($17)
#  Rparent = Rtemp2 ;
#  Ltemp1 = ip     = Rme ;
#  Rtemp1          = (UNFUDGE(
          Ltemp2.child)+1)->value;
        lw      $9, 0($11)
        lw      $8, 4($11)
        lw      $9, 4($9)
        lw      $10, 0($21)
        sw      $8, 0($17)
        sw      $10, 0($18)
        sw      $9, 4($18)
#  *(temp_spine) = temp1+1 ;
#  continue ;
        addu $9, $18, 4
        b       $THREAD
        sw      $9, 0($16)       # delay slot
        .set    reorder

..... NOTE: other cases elided .....

#_____________________________
#  case SPINE UNDERFLOW
        .align          3
$SPINE_UNDERFLOW:
        .loc    2 569
# 566
# 567           /* _____________________________ */
# 568
# 569           case SPINE_UNDERFLOW:
          printf("\n\n spine stack underflow\n");
        la      $4, $$64
        jal     printf
        .loc    2 570
# 570               exit(-1);
        li      $4, -1
        jal     exit

        .set    reorder

# /*_____________________________ */
# /* Supercombinator definition for nfib */
#
# #define fast_mapval(x)  ((x * 4) + 1)

#
        .align          3
# case DO_S6:
        .align          3
        .set    noreorder
$DO_S6:
#   Use_Me;
        lw      $21, 0($16)

#   result = Evaluate(Rme,temp_spine) ;
        addi  $16, -4
                # spine stack parameter element
        jal     $MY_EVAL
        lw      $10, 0($21)       # delay slot


#   if (result.literal < 2 )
```

```
        bge   $11, 2, $S6_A
        addi  $16,  4       # delay slot

#   { result.literal = 1 ;
#     Pop_Spine(1);
#     break ;
        lw    $31, 4($16)
        li    $11, 1
        j     $31
        addu $16, $16, 8
              # pop address  - - delay slot
#   }

$S6_A:
        .set   noreorder
# - — — — — — — — — — -
#   NEWNODEN(8);
        lw    $15, heap_end
        addu $18, $22, 0
        subu $15, $15, 8*8
              # number of cells allocated
        bltu  $18, $15, $S6_OK
        addu $22, $22, 8*8    # delay slot
        .set   reorder
# - - garbage collection - -
        sw    $11, -4($16)
        sw    $23, count
        sw    $22, next_free_node
        li    $4,  8  # number of nodes to allocate
        move $5, $16
        jal   newnoden_continue
        move $18, $2
        lw    $22, next_free_node
        lw    $11, -4($16)
# - — — — — — — -
        .set   noreorder
$S6_OK:

#  load table address into $19
        la    $19, $TABLE
#   Use_Me;
        lw    $21, 0($16)
#   (temp1+0)->value.comb =
          fast_mapval(DO_LIT);
#   (temp1+1)->value.literal = result.literal - 1 ;
        lw    $8, 8 *4($19)
        addu $9, $11, -1
        sw    $8, 0*4($18)
        sw    $9, 1*4($18)
#   (temp1+2)->value.comb =
          fast_mapval(DO_S6);
#   (temp1+3)->value.child =
          FUDGE(temp1+0);
        lw    $9, 41 *4($19)
        sw    $18, 3*4($18)
        sw    $9, 2*4($18)
#   (temp1+4)->value.comb =
          fast_mapval(DO_PLUS);
#   (temp1+5)->value.child =
          FUDGE(temp1+2);
        lw    $8, 9 *4($19)
        addu $9, $18, 2*4
        sw    $8, 4*4($18)
        sw    $9, 5*4($18)
#   (temp1+6)->value.comb =
          fast_mapval(DO_LIT);
#   (temp1+7)->value.literal = result.literal - 2 ;
        lw    $8, 8 *4($19)
        addu $9, $11, -2
        sw    $8, 6*4($18)
        sw    $9, 7*4($18)
#   (temp1+8)->value.comb =
          fast_mapval(DO_S6);
#   (temp1+9)->value.child =
          FUDGE(temp1+6);
        lw    $9, 41 *4($19)
        addu $8, $18, 6*4
        sw    $9, 8*4($18)
        sw    $8, 9*4($18)
#   (temp1+10)->value.child =
          FUDGE(temp1+4);
#   (temp1+11)->value.child =
          FUDGE(temp1+8);
        addu $8, $18, 4*4
        addu $9, $18, 8*4
        sw    $8, 10*4($18)
        sw    $9, 11*4($18)
#   (temp1+12)->value.comb =
          fast_mapval(DO_PLUS);
#   (temp1+13)->value.child =
          fast_mapval(DO_ONE);
        lw    $8, 9 *4($19)
        lw    $9, 13 *4($19)
        sw    $8, 12*4($18)
        sw    $9, 13*4($18)
#   (temp1+14)->value.child =
          FUDGE(temp1+12);
#   (temp1+15)->value.child =
          FUDGE(temp1+10);
        addu $8, $18, 12*4
        addu $9, $18, 10*4
        sw    $8, 14*4($18)
        sw    $9, 15*4($18)


#   Lme.comb = ip.comb = fast_mapval(DO_I);
#   Rme.child = FUDGE(temp1+14);
        lw    $10, 1 *4($19)
        addu $8, $18, 14*4
        sw    $10, -4($21)
#   continue;
        b     $THREAD
        sw    $8, 0($21)
        .set   reorder
# /*_____________________ */
#
#

..... NOTE: other cases elided .....
$BAD_COMBINATOR:
```

```
        .loc    2 572
# 571
# 572           default: printf("\n\n jump to bad
          combinator\n");
        la      $4, $$65
        jal     printf
        .loc    2 573
# 573             exit(-1);
        li      $4, -1
        jal     exit
        .loc    2 574
# 574           }
        b       $MY_BREAK

        .end  Evaluate

#_____________________________
#  mapval routine
        .text
        .align        2
        .file   2 "kernel.c"
        .globl mapval
        .loc    2 584
# 584         { int temp_toknum ;
        .ent    mapval 2
mapval:
        .option       O1
        subu $sp, 8
        .frame        $sp, 8, $31
# 585           temp_toknum = toknum << 2 ;  /* force
          to high bit set */
        sll     $14, $4, 2
# 586           temp_toknum += 1 ;
        lw      $15, $TABLE($14)
 #  leave the shift out for now
 #     srl     $15, $15, 1
 #     or      $15, $15, 0x80000000
        sw      $15, 4($sp)
# 587           return(temp_toknum);}
        move $2, $15
        addu $sp, 8
        j       $31

#_____________________________
#  case DISPATCH CASE STATEMENT - -
          data
        .rdata
        .word$BAD_COMBINATOR + 0x80000000
        .word$BAD_COMBINATOR + 0x80000000
        .word$BAD_COMBINATOR + 0x80000000
        .word$BAD_COMBINATOR + 0x80000000
        .word$BAD_COMBINATOR + 0x80000000
$TABLE:
        .word$BAD_COMBINATOR + 0x80000000
        .word$DO_I + 0x80000000
        .word$DO_K + 0x80000000
        .word$DO_S + 0x80000000
        .word$DO_SPRIME + 0x80000000
        .word$DO_B + 0x80000000
        .word$DO_C + 0x80000000
..... NOTE: other cases elided .....
        .word$BAD_COMBINATOR + 0x80000000
        .word$BAD_COMBINATOR + 0x80000000
        .text
        .end  mapval
1
```

B.5. HEAP.H

```
/* TIGRE implementation:  HEAP.H */
/* Heap management and garbage collection */
/* Uses stop-and-copy garbage collector */

/* (C) Copyright 1989   Philip Koopman Jr.  */
/*  Last update:  4/17/89   */

#include <setjmp.h>
extern jmp_buf env ;
extern int ready_for_longjmp ;  /* true when
          want a setjmp restart */

#define NEWNODE1                                \
  temp1 = next_free_node ;                      \
  next_free_node = temp1 + HEAP_STRIDE ;
          \
  if( temp1 >= heap_end) temp1 =
          newnode_continue(temp_spine);

#define NEWNODE2                        \
  temp1 = next_free_node ;              \
  temp2 = temp1 + HEAP_STRIDE ;         \
  next_free_node = temp1 + 2*HEAP_STRIDE ;
          \
  if( temp2 >= heap_end)                \
   {
          newnode2_continue(temp_spine,&nodea,
          &nodeb); \
         temp1=nodea; temp2=nodeb; }

#define NEWNODE3                        \
  temp1 = next_free_node ;              \
  temp2 = temp1 + HEAP_STRIDE ;         \
  temp3 = temp1 + 2*HEAP_STRIDE ;       \
  next_free_node = temp1 + 3*HEAP_STRIDE ;
          \
  if( temp3 >= heap_end)                \
  {
          newnode3(temp_spine,&nodea,&nodeb,&
          nodec);       \
               temp1=nodea; temp2=nodeb;
          temp3=nodec; }

#define NEWNODEN(x)           \
```

```
 temp1 = next_free_node ;     \
 next_free_node = temp1 + HEAP_STRIDE * x
        ; \
 if( next_free_node >= heap_end) { \
  collect_garbage(temp_spine); \
  temp1 = next_free_node ; \
  next_free_node = temp1 + HEAP_STRIDE *
        x ; \
  if( next_free_node >= heap_end) { \
   printf("Out of heap!"); \
   exit (-1); \
 } };
```

B.6. HEAP.C

```
/* TIGRE implementation:  HEAP.C */
/* Heap management and garbage collection */
/* Uses stop-and-copy garbage collector */

/* (C) Copyright 1989   Philip Koopman Jr.  */
/*  Last update:  4/17/89   */

#include "reduce.h"
#include "heap.h"

  Celltype *root_save ;            /* root of
        evalution tree */
  Celltype *next_free_node ;       /* heap free
        list pointer */
  Celltype *heap_start, *heap_end ;  /*
        boundaries of heap memory */
  Celltype *from_heap, *to_heap ;  /* two heap
        buffers */

  Celltype **spine_start,**spine_end ;  /*
        boundaries of spine memory */
  Celltype **spine_stack ;    /* points to
        topmost element on spine stack */
  Celltype *spine_mem[SPINESIZE+100];    /*
        spine stack storage */
  Celltype **fixup_start ;  /* starting point for
        stack fixup */

  int using_first_heap ; /* true flag with
        first_heap is current */
  int gc ;      /* counts number of garbage
        collections used */


void zapflags()  /* set all flags in heap arrays to
        false */
 { register int i;
   register Celltype *zapptr ;
#if VAX_ASM
    zapptr = from_heap ;
    for (i = 0 ; i <= (HEAPSIZE) ; i++ )
      { zapptr->value.comb = UNMARKED ;
        zapptr += 1 ;
        if(IS_FORWARDED(zapptr->value.child))
              zapptr->value.child =
          UNFORWARD(zapptr->value.child);
        zapptr += HEAP_STRIDE - 1 ;
      }
   zapptr = to_heap ;
   for (i = 0 ; i <= (HEAPSIZE) ; i++ )
     { zapptr->value.comb = UNMARKED ;
       zapptr += 1 ;
       if(IS_FORWARDED(zapptr->value.child))
            zapptr->value.child =
         UNFORWARD(zapptr->value.child);
       zapptr += HEAP_STRIDE - 1 ;
     }
#else
   zapptr = from_heap+1 ;
   for (i = 0 ; i <= (HEAPSIZE) ; i++ )
     { if(IS_FORWARDED(zapptr->value.child))
            zapptr->value.child =
         UNFORWARD(zapptr->value.child);
       zapptr += HEAP_STRIDE ;
     }
   zapptr = to_heap+1 ;
   for (i = 0 ; i <= (HEAPSIZE) ; i++ )
     { if(IS_FORWARDED(zapptr->value.child))
            zapptr->value.child =
         UNFORWARD(zapptr->value.child);
       zapptr += HEAP_STRIDE ;
     }
#endif
 }

void init_heap()  /* init heap memory - - leaves
        first xx nodes free */
{  int i ;  /* loop counter */
#if DEBUG_HEAP
   printf(" init_heap");
#endif
  heap_start = to_heap ;
  heap_end =
        heap_start+(HEAPSIZE*HEAP_STRIDE
        ) ;
  /* set next_free_node halfway through to
        allow room for program */
  next_free_node =
        heap_start+(HEAP_STRIDE*PROGSIZE
        )+1 ;

  spine_start = &spine_mem[2] ;
  spine_end = &spine_mem[SPINESIZE-2] ;
  spine_stack = spine_end ;
  fixup_start = spine_end - 1 ;
  zapflags();
```

```
  for ( i = 0 ; i <= MAX_TOKEN ; i++ )
    { i_counts[i] = 0 ; }
  i_heap = 0 ;
  ready_for_longjmp = FALSE ;
}

Celltype *newnode_continue(top_spine)
  Celltype **top_spine;
{ /* ran out of heap - - do garbage collection */
  register Celltype *mynode ;
  collect_garbage(top_spine);
  mynode = next_free_node ;
  next_free_node = mynode + HEAP_STRIDE ;
  return(mynode) ;
}

Celltype *newnode(top_spine)
 Celltype **top_spine;
{ /* allocate a node unless free node pointer
          goes too far */
  register Celltype *mynode ;
  mynode = next_free_node ;
  next_free_node = mynode + HEAP_STRIDE ;
  if( mynode < heap_end)  return(mynode) ;
  return(newnode_continue(top_spine));
}

void newnode2_continue(top_spine, nodea,
          nodeb)
 Celltype **top_spine, **nodea, **nodeb;
{ register Celltype *temp ;
  /* ran out of heap - - do garbage collection */
  collect_garbage(top_spine);
  temp = next_free_node ;
  *nodea = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodeb = temp ;
  next_free_node = temp + HEAP_STRIDE ;
  return ;
}
void newnode2(top_spine, nodea, nodeb)
  Celltype **top_spine, **nodea, **nodeb;
{ register Celltype *temp ;
  temp = next_free_node ;
  *nodea = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodeb = temp ;
  next_free_node = temp + HEAP_STRIDE ;
  if( temp < heap_end)  return ;
  newnode2_continue(top_spine,nodea,nodeb);
  return;
}

void newnode3_continue(top_spine, nodea,
          nodeb, nodec)
 Celltype **top_spine, **nodea, **nodeb,
          **nodec;
{ register Celltype *temp ;
  /* ran out of heap - - do garbage collection */
  collect_garbage(top_spine);
  temp = next_free_node ;
  *nodea = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodeb = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodec = temp ;
  next_free_node = temp + HEAP_STRIDE ;
  return ;
}
void newnode3(top_spine, nodea, nodeb, nodec)
  Celltype **top_spine, **nodea, **nodeb,
          **nodec;
{ register Celltype *temp ;
  temp = next_free_node ;
  *nodea = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodeb = temp ;
  temp = temp + HEAP_STRIDE ;
  *nodec = temp ;
  next_free_node = temp + HEAP_STRIDE ;
  if( temp < heap_end)  return ;
  newnode3_continue(top_spine,nodea, nodeb,
          nodec) ;
  return;
}


Celltype *newnoden_continue(n, top_spine)
  int n;
  Celltype **top_spine;
{ /* ran out of heap - - do garbage collection */
  register Celltype *mynode ;
  collect_garbage(top_spine);
  mynode = next_free_node ;
  next_free_node = mynode + HEAP_STRIDE *
          n ;
  return(mynode) ;
}


voidcollect_garbage(top_spine)
  Celltype **top_spine;
{ UNIX_REGISTER Celltype from_ptr ; /*
          scratch pointers for the copying */
  UNIX_REGISTER Celltype temp ;
  register Celltype *scanned ;
  UNIX_REGISTER Celltype *unscanned ;
  UNIX_REGISTER Celltype lit_value ;  /*
          holding buffer for LIT compare */
#if DEBUG_HEAP
    printf(" collect_garbage");
#endif
  gc++ ;
  lit_value = do_lit_value ;
#if DEBUG_DUMP
       printf("\n\n *** graph dump %X %X
          ***",heap_start,heap_end);
       zapflags();
       dumpgraph(root_save,0);
```

```
      zapflags();
      printf("\n *** end of graph dump ***\n");
#endif
#if DEBUG_HEAP
  printf("gc#%d “,gc);
#endif
 /* exchange the role of the heaps, since
        to_heap was previously in use */
 temp.value.ptr = to_heap ;
 to_heap = from_heap ;
 from_heap = temp.value.ptr ;
 heap_start = to_heap ;
 heap_end =
        heap_start+(HEAPSIZE*HEAP_STRIDE
        ) ;
 scanned = heap_start + 1 + HEAP_STRIDE ;
 unscanned = scanned + HEAP_STRIDE ;

 /* establish the root node as the base case for
        the to heap */
 scanned->value = root_save->value ;
 (scanned+1)->value = (root_save+1)->value ;
 root_save = scanned ;

 /* scan through the to space until all
        referenced from nodes copied */
 while ( scanned < unscanned )
 {
#if DEBUG_HEAP
   printf("\n node @ %X",scanned);
#endif
     /* — — - copy this section for second half */
   from_ptr.value.ptr = scanned->value.child ;
        /* target of LHS pointer */
#if DEBUG_HEAP
   printf("\n - — LHS =
        %X",from_ptr.value.ptr);
#endif
   if (IS_PTR(from_ptr.value.ptr))
   { /* It's a pointer. */
    from_ptr.value.ptr =
        UNFUDGE(from_ptr.value.ptr);
    temp = *from_ptr.value.ptr ;
    if ( IS_FORWARDED(temp.value.ptr) )
    { /* the target cell is already resident in the
        heap */
#if DEBUG_HEAP
      printf("\n target in heap at
        %X",UNFORWARD(temp.value.ptr));
#endif
      scanned->value.child =
        UNFORWARD(temp.value.ptr);
    }
    else
    { /* the target cell must be moved */
#if DEBUG_HEAP
      printf("\n target moved to
        %X",unscanned);
#endif
      scanned->value.child =
        FUDGE(unscanned);
      *(unscanned) = temp ;
      *(unscanned+1) = *(from_ptr.value.ptr+1) ;
      from_ptr.value.ptr->value.ptr =
        MAKE_FORWARD(
        FUDGE(unscanned));
      unscanned += HEAP_STRIDE ;  /* skip
        the mark word */
    }
   }
     /* - — — - end of copy section */
   else
   { /* It's a combinator */
    if ( scanned->value.comb ==
        lit_value.value.comb )
    { /* if the LHS was a LIT combinator, then
        skip RHS */
#if DEBUG_HEAP
      printf("\n LIT combinator\n");
#endif
      scanned += HEAP_STRIDE;
      continue ;
    }
   }
   /* now do the RHS of the scanned node */
   scanned += 1;
   /* - — - copied from above condition clause */
   from_ptr.value.ptr = scanned->value.child ;
        /* target of LHS pointer */
#if DEBUG_HEAP
   printf("\n - — RHS =
        %X",from_ptr.value.ptr);
#endif
   if (IS_PTR(from_ptr.value.ptr))
   { /* It's a pointer. */
    from_ptr.value.ptr =
        UNFUDGE(from_ptr.value.ptr);
    temp = *from_ptr.value.ptr ;
    if ( IS_FORWARDED(temp.value.ptr) )
    { /* the target cell is already resident in the
        heap */
#if DEBUG_HEAP
      printf("\n target in heap at
        %X",UNFORWARD(temp.value.ptr));
#endif
      scanned->value.child =
        UNFORWARD(temp.value.ptr);
    }
    else
    { /* the target cell must be moved */
#if DEBUG_HEAP
      printf("\n target moved to
        %X",unscanned);
#endif
      scanned->value.child =
        FUDGE(unscanned);
      *(unscanned) = temp ;
      *(unscanned+1) = *(from_ptr.value.ptr+1) ;
      from_ptr.value.ptr->value.ptr =
        MAKE_FORWARD(
```

```
        FUDGE(unscanned)) ;
      unscanned += HEAP_STRIDE ;  /* skip
        the mark word */
    }
   }

   /* - — — - end of copy */
   scanned += (HEAP_STRIDE - 1);  /* skip the
        mark for the next cell */
 } ;

  next_free_node = unscanned ;
  /* do the return stack */
#if DEBUG_HEAP
  printf("rs “);
#endif

  if ( (!ready_for_longjmp)
  && (top_spine != spine_end ))
        fixup_stack(top_spine);
#if DEBUG_DUMP
      printf("\n\n *** graph dump %X %X
        ***",heap_start,heap_end);
      zapflags();
      dumpgraph(root_save,0);
      zapflags();
      printf("\n *** end of graph dump ***\n");
#endif
#if DEBUG_HEAP
  printf("done... \n");
  printf("\nGC  %X used, sp = %X
        “,(next_free_node - heap_start),
          (fixup_start -  top_spine) );
  if ( ready_for_longjmp)  printf (" - - longjmp") ;
#endif

  if( next_free_node < ( heap_end - 256 ) )
   {
    if(ready_for_longjmp)
       { _longjmp(env,1);}
    else {  return ;}
   }
  printf("\n\n ***** heap space overflow
        (burp!) **** \n\n");
  exit(-1);
}
```

```
void fixup_stack(top_spine)
  Celltype **top_spine;
{ UNIX_REGISTER Celltype temp ;
  UNIX_REGISTER Celltype **temp_spine ;

#if DEBUG_HEAP
  printf("\nstack fixup  start:%X
        end:%X\n",fixup_start,top_spine);
#endif

/* fix up the stack for garbage collection */
  temp_spine = top_spine ; /* skip over saved
        pointer to root_save */
  while( temp_spine <= fixup_start )
  { /* copy in forwarded address */
#if DEBUG_HEAP
    printf("\nmoving %X
        (%X)",temp_spine,*temp_spine);
#endif
    temp.value.ptr = *temp_spine ;
    if ( temp.value.ptr < from_heap ) /* check for
        subr call */
      { /* skip a data value */
#if DEBUG_HEAP
       printf("...skip... %X
        %X",*temp_spine,*(temp_spine+1));
#endif
       temp_spine += 2;
       continue ;
      }
    temp = *((*temp_spine)-1) ;  /* get
        forwarding address */
#if 1+DEBUG_HEAP
    if ( ! IS_FORWARDED(temp.value.ptr) )
      printf("\n bad stack forwarding value=%X
        address=%X",
                  temp.value.ptr,*temp_spine);
#endif
    *temp_spine = UNFUDGE(
        UNFORWARD(temp.value.ptr)) + 1 ;
    temp_spine++ ;
  }
#if DEBUG_HEAP
  printf("\ndone\n\n");
#endif
}
```