A Precise Version of the Time Hierarchy for Language I

Amir M. Ben-Amram

August 24, 1998

1 Introduction

This report repeats the proof of the hierarchy theorem for language `I` in the paper [1] (henceforth *the paper*) in a sufficient level of detail that the constants involved can be evaluated, resulting in the following

Proposition 1.1 *Let $T(n)$ be a time-constructible function such that the program constructing $T(n)$ runs in time bounded by $aT(n)$.*
Then $\mathrm{TIME}^{\mathtt{I}}((a+124)T(n)+105n) \setminus \mathrm{TIME}^{\mathtt{I}}(T(n))$ *is non-empty.*

Proposition 1.2 *For all integer* $a \geq 1$, $\mathrm{TIME}^{\mathtt{I}}((126a{+}103)n) \setminus \mathrm{TIME}^{\mathtt{I}}(an)$ *is non-empty.*

The proof of the hierarchy theorems hinges on the construction of a program `diag` whose running time should be analyzed. The program includes two iterative parts: computation of the time bound and timed interpretation of a program. The main effort was in coding these fragments in full and analyzing their running time. We remind the structure of the program `diag` used in the proof:

```
read X;
Timebound := nil^{⌈a⌉·|X|};
X := tu(X,X,Timebound);
if hd X then X := nil else X := (nil.nil);
write X
```

Actually this program is used for the linear-time hierarchy theorem. Replacing the computation of `Timebound` by a code for computing $\mathtt{nil}^{T(|\mathtt{X}|)}$ gives the general result.

2 Implementation and Analysis of STEP

In order to evaluate the running time of non-trivial program segments, I have coded them in full, tested them for correctness and analyzed their running time using an auxiliary program. All the programming has been carried out under the CMU Common Lisp system.

The running time of a single application of STEP depends on the rule chosen, and is the sum of the time spent on choosing the rule and the time of executing the rule itself: we refer to these quantities as *branching time* and *rule time.* The worst-case time of STEP is the maximum sum of branching time and rule time over the different rules.

2.1 Encoding of multiple variables and atoms

In order to code the program in the `I` language we have to decide on a representation of the four variables and the many special atoms in terms of the single variable `X` and the single atom `nil`.
I chose to represent the variables of STEP by setting `X` to

`(Cd.(Stk.(Cntr.Val)))`

Thus `hd Cd` is accessed by `hd hd X`, etc.
The special atoms that have to be encoded are the following:

```
; := do_assgn if do_if while do_while X quote hd do_hd
tl do_tl cons do_cons
```

The fact that I work with Common Lisp forced me to rename the atoms `;` and `:=` so in the program texts they go by the names `$` and `assign`.

The atoms can be divided in two groups. One group is atoms that appear at the start of a list: these are the atoms that are part of the `I` concrete syntax, for example `while` appears in the list `(while E C)`. The other atoms appear only on their own: these are the atoms used internally by the step macro, for example `do_while`.

The first distinction that the top-level of STEP has to do is between these two cases. In order to economize on time, I have not used the general transformation used to encode arbitrary structures over multiple atoms in the paper. Instead, I have made it easy to distinguish between the two groups of atoms by encoding all stand-alone atoms as values whose head is `nil`. Since all special atoms are encoded as non-`nil` values, it is now possible to decide which case is before us by looking at `hd hd Cd`. Except for that, the particular encoding is quite arbitrary, and was chosen so that rules that

do_if	(nil nil)
do_cons	(nil nil nil)
do_while	(nil (nil))
do_tl	(nil (nil) nil)
do_hd	(nil (nil nil))
X	(nil (nil nil) nil)
do_assgn	(nil ((nil) nil) nil)
if	(nil)
quote	(nil nil)
:=	((nil))
;	((nil) nil)
cons	(((nil)))
while	(((nil)) nil)
tl	(((nil) nil))
hd	(((nil) nil) nil))

Figure 1: Encoding for atoms, written in list notation.

take longer time to execute will be faster to recognize, thus minimizing the worst-case time of STEP.

2.2 Implementation and Analysis

Given the encodings, writing I code for the STEP macro is stright-forward. This code appears in file `step.i` (a printout of all the files mentioned is enclosed). For ease of development and testing, the top level has been coded separately first in file `top.i`. Testing programs `test-top.cl` and `test-step.cl` (written in Common Lisp) where used to verify the correctness of these programs. A *time analysis program* (`time.cl`) was written to compute the running time of individual rules and of the whole macro. The program operates according to a simple recursive formula for the time complexity of an expression. For readability of the output, it determines the identity of the rules by comparing the full code (from `step.i`) with the code in `top.i` that only gives the rule name instead of the rule. The final results follow: they should be self-explanatory.

```
(('HD 48) ('TL 48) ('WHILE 48) ('CONS 55) ('$ 46) (':= 52)
('QUOTE 44) ('IF 49) ('DO_ASSGN 54) ('X 58) ('DO_HD 53)
('DO_TL 53) ("DO_WHILE:t" 59) ("DO_WHILE:f" 53) ('DO_CONS 52)
("DO_IF:t" 52) ("DO_IF:f" 53) ('NIL 14))
```

These figures include the branching time. The worst-case time of STEP is thus 59.

3 Running time of the interpretation loop

The main loop in program `diag` is the interpretation loop, at the heart of the timed interpreter `tu`. This loop is written in the paper as follows:

```
while Cd and Cntr do STEP
```

In the sake of conciseness, we change this to

```
while Cntr do STEP
```

Macro STEP has a rule for the case of empty Code stack, in which we set `Cntr` to `nil`; this preserves the meaning of the loop, except for the limit case in which `Cd` becomes empty at the very step where `Cntr` goes down to zero. In this case our program will conclude incorrectly that the interpreted program has outrun the time limit. Thus, the condition $time_{\mathtt{p}}(\mathtt{d}) \leq n$ at the specification of `tu` will change to $time_{\mathtt{p}}(\mathtt{d}) < n$. But this can be easily taken care of by making `Cntr` larger by one at the start.

Let $T_{\mathrm{STEP}}(n)$ be the time spent inside the interpretation loop (i.e., not counting loop management). We now bound $T_{\mathrm{STEP}}(n)$. The paper gives an analysis which shows, that the number of applications of STEP when interpreting program p with a time buond of n is bounded by $3n + |\mathrm{p}|$. We now make the analysis more precise. The bound is obtained by dividing the iterations of steps in two groups, the marked rules and the unmarked rules, where every marked rule decreases `Cntr`. If N_1 is the number of applications of marked rules and N_2 that of unmarked rules, we have $N_1 \leq n$. To bound N_2 we consider the code stack. We note that only marked rules pop the stack. It follows that we can bound N_2 in terms of N_1 and R, the number of values left at the stack when `Cntr` goes to zero. The paper gives the rough estimate $N_2 \leq 2N_1 + R$, where the coefficient of 2 is due to the fact that some marked rules pop two elements of the stack. However, a careful study of the rules reveals that every rule that pops two values is matched by a preceding rule that pushes at least two; the doubling is thus redundant, and we have $N_2 \leq N_1 + R$, If we let t_m stand for the maximal running time of a marked rule, and t_u for that of an unmarked rule,we obtain $T_{\mathrm{STEP}}(n) \leq nt_m + (n + R)t_u$.

The fact, that we increased the value of `Cntr` by one, does not have to be taken into account — the reader may verify that our bound is loose enough to absorb this.

The value of R has been bounded in the paper by $|\mathtt{p}|$, based on the observation that the stack always contains a set of different fragments of the program, so their number is bounded by its size. Actually, $|\mathtt{p}|$ is an overestimation because before pushing the stack the program always verifies that `hd hd Cd` is non-`nil`; and the fragment that is pushed is taken from `tl hd Cd`. So the size of the fragments decreases by at least 4 as we go up the stack, and we obtain $R \le \frac{1}{4}|\mathtt{p}|$. The values of t_m and t_u are 59 and 55 respectively (see the list at the end of the last section), so

$$T_{\mathrm{STEP}}(n) \le 114n + \frac{55}{4}|\mathtt{p}| \le 114n + 14|\mathtt{p}|\,. \qquad (1)$$

4 Computing `Timebound`

The computation of `Timebound` is made using the program for computing the size of a tree that is presented in the paper. Complete `I` code appears in file `tree-size.i`, which is made to run under Common Lisp and has been tested. The program starts with the input tree `d` in the variable `X` and ends with `X=(nil.`$|$`d`$|$`.d)` so that both the tree and the size are available for further processing.

Running time: consider the two branches inside the `while` loop. The first branch does not increment the size, but moves an internal node of the tree into the rightmost path. No node is ever moved out of the rightmost path so the number of such iterations is bounded by the number of internal nodes less one, i.e., $(|\mathtt{d}| - 1)/2 - 1 = (|\mathtt{d}| - 3)/2$.

The second branch increments the size by two, while reducing the tree, so it is clearly executed $(|\mathtt{d}| - 1)/2$ times.

We obtain the cost of each of the branches, as well as the non-iterative part, by applying the time analysis program. The results appear as annotations in the source code. An easy calculation (omitted here) gives the final result, which we describe as a function of $n = |\mathtt{d}|$: $20.5 \cdot n - 30.5$, which we round up to bound $21n - 33$.

So far we have computed $|\mathtt{d}|$. Program `diag` calls for computing $a \cdot |\mathtt{d}|$. To obtain this result we just have to change the initial value of the counter from 1 to a and increment the counter by $2a$ instead of 2 each time. Adapting our calculations to this change gives the following result. We denote the time for this part by $t_{\mathrm{bound}}(n, a)$:

$$t_{\mathrm{bound}}(n, a) \le (19 + 2a)n - 2a - 31\,. \qquad (2)$$

```
read X;
Compute Timebound;
X := (cons (cons (tl tl X) nil)
     (cons nil (cons (cons nil (tl X)) (tl tl X))));
while (hd tl tl X) do STEP;
if (hd X) then        (* not finished on time *)
  X := nil
else       (* finished on time, invert result *)
  if (tl tl tl X) then
    X := nil
  else
    X := X;
write X
```

Figure 2: Program `diag`.

5 Completion of the program

We now complete program `diag`. In contrast with the paper, I have not written `tu` separately and used it as a macro but written program `diag` directly, using the pieces we already have. The resulting program appears in Figure 2. The assignment after computation of `Timebound` takes the contents of `X`, that include the time bound and the original input (see last section), and creates the structure that STEP expects (`Cd.Stk.Cntr.Val`).

Running time. Let n denote the size of the input. The time for computing `Timebound` is given by (2). The non-iterative parts before and after the main loop take 27 time units. Consider the main loop: the time spent in all the iterations of STEP put together is given by substituting an for the time bound in (1) to obtain $114an + 14n$. From the reasoning leading to that bound, we can also deduce that the *number* of iterations is bounded by $2an + 14n$. Add one for the execution of the `while` command that fails, multiply by the time of the `while` command (5 units), and we obtain the expression $10an + 70n + 5$. Putting the pieces together we obtain:

$$(19 + 2a)n - 2a - 31 + 27 + 114an + 14n + 10an + 70n + 5 < (126a + 103)n$$

We conclude that $Acc(\texttt{diag}) \in \mathrm{LIN}^{\mathtt{I}}(126a + 103)$. The diagonal argument shows that it cannot be in $\mathrm{LIN}^{\mathtt{I}}(a)$, proving Proposition 1.2. For the general case of a time-constructible bound $T(n)$, we use the program we gave to compute the input size (as for $a = 1$), followed the program that computes $T(n)$. Adjusting the calculations gives Proposition 1.1.

References

[1] A. M. Ben-Amram and N. D. Jones, *Computational complexity via programming languages: constant factors do matter*, submitted for publication, 1998.

top.i

```
;
; STEP macro - top level only
; here Cd stands for the code stack.
;
;(defun step (Cd)
(if Cd
    (if (hd (hd Cd))
        (if (hd (hd (hd Cd)))
            (if (hd (hd (hd (hd Cd))))
            (if (tl (hd (hd (hd Cd))))
                    (if (tl (hd (hd Cd)))
                        'hd
                        'tl
                    )
                    (if (tl (hd (hd Cd)))
                    'while
                    'cons
                    )
                )
                (if (tl (hd (hd Cd)))
                    '$
                    ':=
                )
            )
            (if (tl (hd (hd Cd)))
            'quote
            'if
            )
        )
;
;(hd (hd Cd)) = nil, so Fst is the code of an atom
;
        (if (hd (tl (hd Cd)))
            (if (tl (hd (tl (hd Cd))))
                (if (tl (tl (hd Cd)))
                (if (hd (hd (tl (hd Cd))))
                    'do_assgn
                        'X
                )
                    'do_hd
                )
                (if (tl (tl (hd Cd)))
                    'do_tl
                    'do_while
                )
```

```
      )
      (if (tl (tl (hd Cd)))
        'do_cons
        'do_if
      )
    )
  )
  'nil          ;nil case
)
;) ;defun
```

step.i

```
;
; STEP macro
;
; with representation:
; X = (Cd . ( Stk . ( Cntr . Val ) ) )
; for atom encoding see file test-step.cl
;
;(defun step (X)
(if (hd X)      ;if there is Cd
(if (hd (hd (hd X)))
    (if (hd (hd (hd (hd X))))
        (if (hd (hd (hd (hd (hd X)))))
                (if (tl (hd (hd (hd (hd X)))))
                (if (tl (hd (hd (hd X))))
                    ;hd
                    (:= (cons (cons (tl (hd (hd X))) (cons (quote do_hd)
                        (tl (hd X)))) (tl X)))
                    ;tl
                    (:= (cons (cons (tl (hd (hd X))) (cons (quote do_tl)
                        (tl (hd X)))) (tl X)))
                )
                (if (tl (hd (hd (hd X))))
                    ;while
                    (:= (cons (cons (hd (tl (hd (hd X)))) (cons (quote do_while)
                        (hd X))) (tl X)))
                    ;cons
                    (:= (cons (cons (hd (tl (hd (hd X)))) (cons
                        (tl (tl (hd (hd X)))) (cons (quote do_cons) (tl (hd X))
                        ))) (tl X)))
                )
            )
            (if (tl (hd (hd (hd X))))
               ;semicolon
               (:= (cons (cons (hd (tl (hd (hd X)))) (cons (tl (tl (hd (hd X))))
                   (tl (hd X)))) (tl X)))
               ;:=
               (:= (cons (cons (tl (hd (hd X))) (cons (quote do_assign)
                   (tl (hd X)))) (cons (hd (tl X)) (cons
                   (hd (tl (tl X))) (tl (tl (tl X)))))))
            )
        )
        (if (tl (hd (hd (hd X))))
            ;quote
            (:= (cons (tl (hd X)) (cons (cons (tl (hd (hd X)))
                (hd (tl X))) (cons (tl (hd (tl (tl X))))
                (tl (tl (tl X)))))))
            ;if
            (:= (cons (cons (hd (tl (hd (hd X)))) (cons (quote do_if) (cons
                (hd (tl (tl (hd (hd X))))) (cons (tl (tl (tl (hd (hd X)))))
                (tl (hd X)))))) (tl X)))
        )
    )
;
;(hd (hd (hd X))) = nil, so (hd (hd X)) is the code of an atom
;
    (if (hd (tl (hd (hd X))))
        (if (tl (hd (tl (hd (hd X)))))
            (if (tl (tl (hd (hd X))))
                (if (hd (hd (tl (hd (hd X)))))
                    ;do_assgn
                    (:= (cons (tl (hd X)) (cons (tl (hd (tl X))) (cons
                        (tl (hd (tl (tl X)))) (hd (hd (tl X)))))))
                    ;X
                    (:= (cons (tl (hd X)) (cons (cons (tl (tl (tl X)))
                        (hd (tl X))) (cons (tl (hd (tl (tl X))))
                        (tl (tl (tl X)))))))
                )
                ;do_hd
                (:= (cons (tl (hd X)) (cons (cons (hd (hd (hd (tl X))))
                    (tl (hd (tl X)))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
            )
            (if (tl (tl (hd (hd X))))
                ;do_tl
                (:= (cons (tl (hd X)) (cons (cons (tl (hd (hd (tl X))))
                    (tl (hd (tl X)))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
                ;do_while
                (if (hd (hd (tl X))) (:= (cons (cons (tl (tl (hd (tl (hd X)))))
                    (tl (hd X))) (cons (tl (hd (tl X)))
                    (cons (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
                    (:= (cons (tl (tl (hd X)))
                    (cons (tl (hd (tl X))) (cons (tl (hd (tl (tl X))))
                    (tl (tl (tl X)))))))
                )
            )
        )
        (if (tl (tl (hd (hd X))))
            ;do_cons
            (:= (cons (tl (hd X)) (cons (cons (cons (hd (tl (hd (tl X))))
                (hd (hd (tl X)))) (tl (tl (hd (tl X)))))
                (cons (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
            ;do_if
            (if (hd (hd (tl X))) (:= (cons (cons (hd (tl (hd X)))
                (tl (tl (tl (hd X))))) (cons (tl (hd (tl X))) (cons
                (tl (hd (tl (tl X)))) (tl (tl (tl X)))))))
                (:= (cons (cons (hd (tl (tl (hd X))))
                    (tl (tl (tl (hd X))))) (cons (tl (hd (tl X))) (cons
                    (tl (hd (tl (tl X)))) (tl (tl (tl X))))))))
        )
    )
)
;if Cd is empty
(:= (cons (quote nil) (cons (quote nil) (cons (quote nil) (tl (tl (tl X)))))))
)
;) ;defun
```

test-top.cl

```
;
;
;  testing step-top
;
;

(defmacro hd (expr) `(car ,expr))
(defmacro tl (expr) `(cdr ,expr))
(defmacro := (x) x)

(defun test ()
(setq do_if    '(nil nil))
(setq do_cons  '(nil nil nil))
(setq do_while '(nil (nil)))
(setq do_tl    '(nil (nil) nil))
(setq do_hd    '(nil (nil nil)))
(setq X        '(nil (nil nil) nil))
(setq do_assgn '(nil ((nil) nil) nil))
(setq if       '(nil))
(setq quote    '(nil nil))
(setq assign   '((nil)))
(setq $        '((nil) nil))
(setq cons     '(((nil))))
(setq while    '(((nil)) nil))
(setq tl       '(((nil) nil)))
(setq hd       '(((nil) nil) nil))

(print (step nil))
(print (step (cons do_if 'Cd)))
(print (step (cons do_cons 'Cd)))
(print (step (cons do_while 'Cd)))
(print (step (cons do_tl 'Cd)))
(print (step (cons do_hd 'Cd)))
(print (step (cons X 'Cd)))
(print (step (cons do_assgn 'Cd)))
(print (step (cons (cons if nil) 'Cd)))
(print (step (cons (cons quote nil) 'Cd)))
(print (step (cons (cons assign nil) 'Cd)))
(print (step (cons (cons $ nil) 'Cd)))
(print (step (cons (cons cons nil) 'Cd)))
(print (step (cons (cons while nil) 'Cd)))
(print (step (cons (cons tl nil) 'Cd)))
(print (step (cons (cons hd nil) 'Cd)))
'-----
)
```

test-step.cl

```
;
;
;  testing STEP
;  contains the functions:
;       init - define atom encoding
;       test - try out every rule and compare with the right results.
;              the results are written in capitals (LISP does not distinguish).
;              returns a Boolean value denoting the success of the test
;

(defmacro hd (expr) `(car ,expr))
(defmacro tl (expr) `(cdr ,expr))
(defmacro := (x) x)

(defun init ()
(setq do_if    '(nil nil))
(setq do_cons  '(nil nil nil))
(setq do_while '(nil (nil)))
(setq do_tl    '(nil (nil) nil))
(setq do_hd    '(nil (nil nil)))
(setq X        '(nil (nil nil) nil))
(setq do_assgn '(nil ((nil) nil) nil))
(setq if       '(nil))
(setq quote    '(nil nil))
(setq assign   '((nil)))
(setq $        '((nil) nil))
(setq cons     '(((nil))))
(setq while    '(((nil)) nil))
(setq tl       '((((nil)) nil)))
(setq hd       '((((nil)) nil) nil))
)

(defun test ()
(and
;rule 0: nil
(equal
(print (step (list* nil 'Stk 'Cntr 'Val)))
'(NIL NIL NIL . VAL)
)
;rule 1: semicolon
(equal
(print (step (list* (list* (list* $ 'C1 'C2) 'Cd) 'Stk 'Cntr 'Val)))
'((C1 C2 . CD) STK CNTR . VAL)
)
;rule 2: assign
```

```
(equal
(print (step (list* (list* (list* assign 'E) 'Cd) 'Stk 'Cntr 'Val)))
'((E DO_ASSIGN . CD) STK CNTR . VAL)
)
;rule 3: do_assgn
(equal
(print (step (list* (list* do_assgn 'Cd) (list* 'w 'Stk) '(nil) 'Val)))
'(CD STK NIL . W)
)
;rule 4: if
(equal
(print (step (list* (list* (list* if 'E 'C1 'C2) 'Cd) 'Stk 'Cntr 'Val)))
'((E DO_IF C1 C2 . CD) STK CNTR . VAL)
)
;rule 5: do_if (2 cases)
(equal
(print (step (list* (list* do_if 'C1 'C2 'Cd) (list* '(t.u) 'Stk) '(nil) 'Val)))
'((C1 . CD) STK NIL . VAL)
)
(equal
(print (step (list* (list* do_if 'C1 'C2 'Cd) (list*    nil 'Stk) '(nil) 'Val)))
'((C2 . CD) STK NIL . VAL)
)
;rule 6: while
(equal
(print (step (list* (list* (list* while 'E 'C) 'Cd) 'Stk 'Cntr 'Val)))
(list* (list* 'E 'DO_WHILE (list* WHILE 'E 'C) 'CD) 'STK 'CNTR 'VAL)
)
;rule 7: do_while (2 cases)
(equal
(print (step (list* (list* do_while (list* while 'E 'C) 'Cd) (list* '(t.u) 'Stk)
             '(nil) 'Val)))
(list* (list* 'C (list* WHILE 'E 'C) 'CD) 'STK NIL 'VAL)
)
(equal
(print (step (list* (list* do_while (list* while 'E 'C) 'Cd) (list*  nil 'Stk)
             '(nil) 'Val)))
'(CD STK NIL . VAL)
)
;rule 8: X
(equal
(print (step (list* (list* X 'Cd) 'Stk '(nil) 'Val)))
'(CD (VAL . STK) NIL . VAL)
)
;rule 9: quote
(equal
(print (step (list* (list* (list* quote 'd) 'Cd) 'Stk '(nil) 'Val)))
'(CD (D . STK) NIL . VAL)
)
```

```
;rule 10: hd
(equal
(print (step (list* (list* (list* hd 'E) 'Cd) 'Stk 'Cntr 'Val)))
'((E DO_HD . CD) STK CNTR . VAL)
)
;rule 11: do_hd
(equal
(print (step (list* (list* do_hd 'Cd) (list* '(t . u) 'Stk) '(nil) 'Val)))
'(CD (T . STK) NIL . VAL)
)
;rule 12: tl
(equal
(print (step (list* (list* (list* tl 'E) 'Cd) 'Stk 'Cntr 'Val)))
'((E DO_TL . CD) STK CNTR . VAL)
)
;rule 13: do_tl
(equal
(print (step (list* (list* do_tl 'Cd) (list* '(t . u) 'Stk) '(nil) 'Val)))
'(CD (U . STK) NIL . VAL)
)
;rule 14: cons
(equal
(print (step (list* (list* (list* cons 'E1 'E2) 'Cd) 'Stk 'Cntr 'Val)))
'((E1 E2 DO_CONS . CD) STK CNTR . VAL)
)
;rule 15: do_cons
(equal
(print (step (list* (list* do_cons 'Cd) (list* 'u 't 'Stk) '(nil) 'Val)))
'(CD ((T . U) . STK) NIL . VAL)
)
))
```

time.cl

```
;
; functions to compute timing for STEP macro (and non-iterative I commands
; in general).
;
; functions:
; time - the general case.
; timebranch - to time branches of a piece of code with if's and $'s.
; timestep, timerules - for handling the STEP macro:
; the input is the bodies of FIRST: step (see step.i) and
; SECOND: step top level ; (see step-top.i).
;
; timestep will compute for every rule
; how much time it takes if this rule is chosen.
; the output consists of a list of 2-lists: (rule name, time).
; the "rule name" is, for example, ':= for the rule that handles this atom;
; for rules that have cases we have names as "DO_IF:t" and "DO_IF:f" with
; the obvious meaning.
;
; timerules does a similar computation but without the time that the
; choosing of the rule ("top level branching") takes, and without
; distinguishing subcases.
;
; NOTE - the arguments are commands written using the usual multi-atom concrete
; syntax, not the "encoded" syntax that my STEP interprets.  So it can be
; applied to the source of STEP.
;
; if the source of step is in file step.i (without a "defun", just a lisp form)
; you can use the following commands to time it:
; (setq step (read (open "step.i")))
; (setq top  (read (open "top.i")))
; (timerules step top)
; (timestep  step top)
;

(defun check (C D)      ;compare if condition from step and from top.
    (if (eql D 'Cd)
        (equal C '(hd X))
        (if (eql (car C) (car D))
            (check (cadr C) (cadr D))
            nil
        )
    )
)

(defun timestep (C D)  ; C and D are commands. C from step and D from top.
    (if (eq (car D) 'quote)
        (if (eq (car C) 'if)    ;rule with subcases
          (list (list (concatenate 'string (string (cadr D)) ":t")
                      (1+ (+ (time (cadr C)) (time (caddr C)))))
                (list (concatenate 'string (string (cadr D)) ":f")
                      (1+ (+ (time (cadr C)) (time (cadddr C))))))
          (list (list D (time C)))
        )
        (if (check (cadr C) (cadr D))   ;safety check
          (let ((IfCost (1+ (time (cadr C)))))
            (append (mapcar #'(lambda (p) (list (car p) (+ IfCost (cadr p))))
                      (timestep (caddr C) (caddr D)))
                    (mapcar #'(lambda (p) (list (car p) (+ IfCost (cadr p))))
                      (timestep (cadddr C) (cadddr D)))))
          (progn
            (princ "inconsistency between top and step at: ")
            (print D)
            nil
          )
        )
    )
)

(defun timerules (C D)
    (if (eq (car D) 'quote)
        (list (list D (time C)))
        (if (check (cadr C) (cadr D))   ;safety check
            (append (timerules (caddr C) (caddr D))
                    (timerules (cadddr C) (cadddr D)))
          (progn
            (princ "inconsistency between top and step at: ")
            (print D)
            nil
          )
        )
    )
)

(defun timebranch (C)
    (case (car C)
        (if  (append (timerules (caddr C))
                     (timerules (cadddr C))))
        ($   (append (timerules (cadr C))
                     (timerules (caddr C))))
        (:=  (list (1+ (time (cadr C)))))
    )
)

(defun time (C)
  (if (eq C 'X) 1
    (case (car C)
        (hd   (1+ (time (cadr C))))
        (tl   (1+ (time (cadr C))))
        (cons (1+ (+ (time (cadr C)) (time (caddr C)))))
        (quote 1)
        (if   (1+ (+ (time (cadr C)) (max (time (caddr C)) (time (cadddr C))))))
        (:=   (1+ (time (cadr C))))
        ($    (+  (time (cadr C)) (time (caddr C))))
    )
  )
)
```

tree-size.i

```
;       tree-size.i
;      =============
;
; this is the tree-size computation program.
; the two variables of the program (X,Y) are represented here as X = (X.Y.Z)
; where Z is a copy of X for use by the rest of the program!
; this is however taken care of by the first assignment so the input to
; the program is simply X. The output has the form (nil.Y.Z).
;

; I language definitions (more complex than in step because we need
; imperative features here).

(defmacro hd (expr) `(car ,expr))
(defmacro tl (expr) `(cdr ,expr))
(defmacro := (expr) `(setq X ,expr))
(defmacro $ (C1 C2) `(progn ,C1 ,C2 X))
(defmacro while (test body) `(do () ((not ,test) X) ,body))


(defun size (X)
($
  (:= (cons X (cons (quote (nil)) X)))                 ;TIME = 6
  (while (hd X)                                        ;TIME = 3
    (if (hd (hd X))
        (:= (cons (cons (hd (hd (hd X))) (cons (tl (hd (hd X))) (tl (hd X))))
          (tl X)))                                     ;TIME = 21
        (:= (cons (tl (hd X))
            (cons (cons (quote nil) (cons (quote nil) (hd (tl X))))
            (tl (tl X))                                ;TIME = 20
        )))
    )
  )
)
)
```