Troisième Colloque de Lille
Les arbres en algèbre et en programmation
16,17,18 février 1978

COMPUTER-TREES AND THEIR PROGRAMMING

B. Buchberger
Institut für Mathematik
Johannes Kepler Universität
A-4045 LINZ

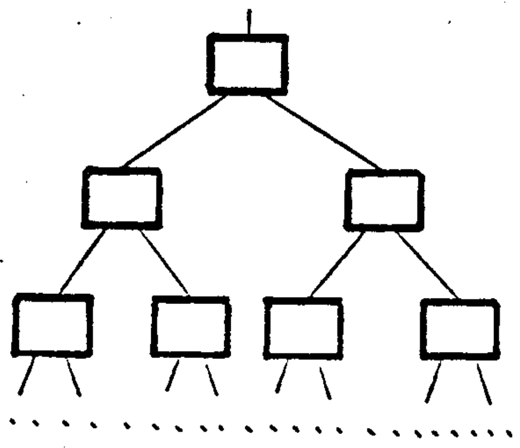
In this paper we would like to contribute to the further investigation of non-standard computer structures, in particular, multiprocessor systems. In the last few years much research has been devoted to the development of multiprocessor systems. In the book of Enslow /1/ the state of the art by 1974 is reviewed. Meanwhile many new ideas have evolved, that are heavily influenced, especially, by the rapid progress in microprocessor technology (see the proceedings of the various conferences on parallel processing and microprocessors, for instance /2/, /3/, /4/, /5/). Among the many different contributions the theoretical work of Albrecht /6/, Glushkov et al. /7/, Schwenkel /8/, and recent proposals of Gostelow /9/, Händler /10/ and Vorgrimmler and Gemmar /11/ seem to be closest to the approach we want to present in this paper. In favour of our approach we would like to emphasize its simplicity and modularity and the easy availability of test hardware. On the other hand, we are aware of the preliminary character of what we can present so far.

We first shall introduce a hardware structure called computer tree (Section 1) and a corresponding (type of) programming language (Section 2). We then shall present a number of programming examples designed to explore the possibilities contained in the approach (Section 3).

The style of the presentation will be non-technical in order to emphasize the underlying concepts. The two essential points in the paper are the Address Modification Rule in Section 2 and Remark 1 in Section 3.

1. Computer-Trees

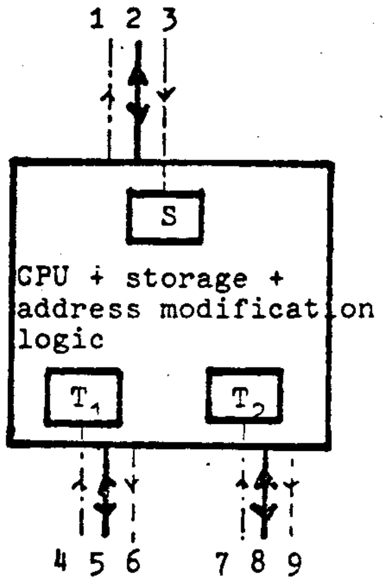
We define a computer-tree to be a hardware structure of the following type:



Every node of the tree denotes a (micro-)computer with its own central processing unit and its own storage. The number of nodes in the tree is potentially infinite. (Of course, a practical implementation has to cut off the tree at some level.)
Every computer of the tree has access to its own storage and to the storage of its left and right son (The type of hardware realization of the access, theoretically, is of no concern. In a practical implementation, direct access is ideal). Accessing the left and right son, a certain type of address modification has to take place, that will be explained in Section 2. Furthermore, we suppose that part of the storage in every node is "private", i.e. cannot be accessed by the father of the node. The private memory will contain the program.

Every computer of the tree must have the possibility to exchange synchronizing information with its neighbours. For this we choose the following mechanism (In fact, a number of other realisations, for instance, an iterrupt system, would work as well without affecting the overall objectives): Every computer of the tree must have three sensorbits S, T_1 and T_2, which it can only sense (not set nor reset). They may be set and reset only by its father, its left son and its right son, respectively.

Thus, every computer C of the tree is a module of the following type:



1.....line enabling C to set and reset T_1 (or T_2) in the father of C,
2.....data and address line for enabling the father of C to access the storage of C,
3.....line enabling the father of C to set and reset S,
4(7)..line enabling the left (right) son of C to set and reset T_1(T_2),
5(8)..data and address line enabling C to access the storage of its left (right) son,
6(9)..line enabling C to set and reset S in its left (right) son.

The hardware of C is supposed to mutually lock the access to the storage of C by its own central processing unit and by its father. The type of hardware-lock is not essential for the overall design, because, as we shall see, by the appropriate use of the sensor bits one can controll the concurrent access to the storage by software means.

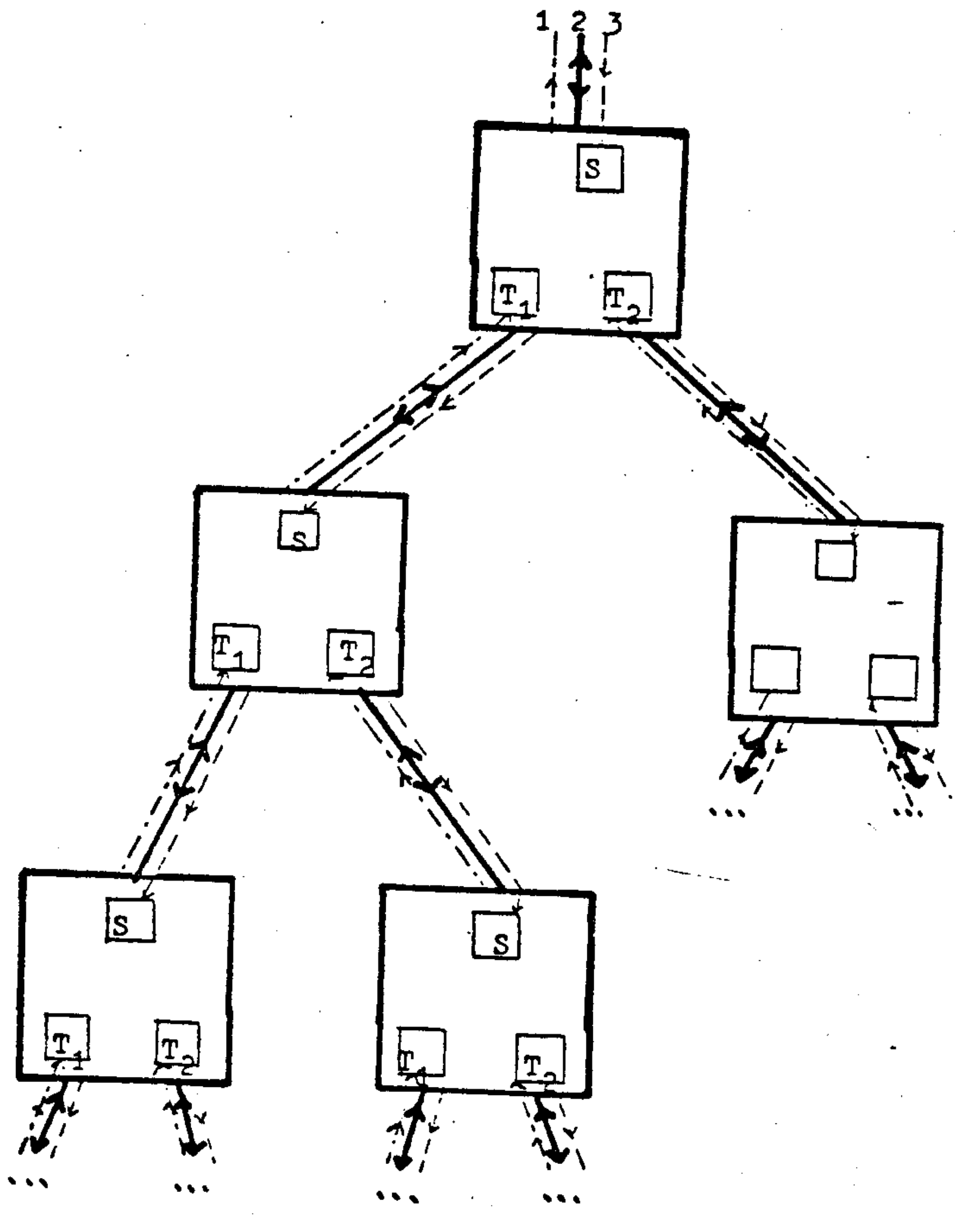
It is clear how to connect an arbitrary number of modules of the above type to form trees: connect lines 4,5,6 (and 7,8,9) of one module to lines 1,2,3 of some other module.

(Though, in this paper, we are not concerned with hardware realizations we remark that it is very easy to realize hardware modules of the above type by combining

a standard microprocessor,
a standard random access memory, and
a suitable interface (containing the three sensor
bits and the address modification logic).

Test modules have been realized, see /12/. In view of the development of microprocessor technology the availability of 2^{15} and more modules in one tree would be realistic).

Summarizing, a computer tree looks like this:



Note, that at the top of the tree three lines of the type 1,2,3 are free. They can be used in the following way (see also mode of operation, section 2):

Line 2 ... for input and output
Line 3 ... for starting a computation
Line 1 ... for deciding whether a computation has terminated

2. A programming language for computer-trees:

Since the basic principle of our approach is language-independent we suppose the computer tree to be programmed in an ALGOL-like language with two additional features:

1. Every variable x is available in three issues x,x',x'' with the following sementical interpretation:

   Basic Address Modification Rule:

   A computer module C in the tree, by means of variables (addresses) of the types x,x' and x'', has access to its own storage,to the storage of its left son C' and to the storage of its right son C'', respectively. Furthermore, if module C addresses a storage region of his left son C' by variable x' then module C' may address the same region by variable x, and analogously for C and its right son C''.

2. Sensor instructions:

   if S then ...;
   if T_1then ...;
   if T_2then ...;
   (the semantics of these instructions is straightforward)

   U:= true; U:= false;
   (by these instructions a module C may set and reset the sensor bit T_1 or T_2 of its father depending on whether C is the left or right son of its father)

   V_1:= true; V_1:= false;
   V_2:= true; V_2:= false;
   (set and reset the sensor bit S in the left and right son, respectively).

Mode of operation: The normal mode of operation for a computer tree consists of the following steps:

1. Load the same program to (the private memory of) all modules in the tree.

2. Reset the sensor bits of all modules to false.
3. Load the data to the top module via line 2.
4. Simultaneously start all modules.
5. Set the sensor bit S of the top module via line 3 to true (Computation starts).
6. Wait until the top module sends the signal true via line 1. (Computation terminates).

It will soon be clear (Section 3) how this mode of operation combines with an appropriate program structure to yield useful computations.

For this starting routine some additional hardware would be necessary. For instance, for 1. we would need additional external access lines to the private memory. However, we also can avoid this by calling a loading routine that works itself according to the basic philosophy of computer tree programming described below, see Example 8 in Section 3.

3. Programming Example:

Example 1:
Consider the tautology problem which is recursively defined by

$$\begin{aligned} \text{taut}\ (t,n) \Longleftrightarrow\ & n=0 \wedge \text{true}\ (t) \vee \\ & n\geq 1 \wedge \text{taut}\ (\text{subst}(t,n,1),n-1) \wedge \\ & \qquad \text{taut}\ (\text{subst}(t,n,0),n-1), \end{aligned}$$

where we used the following abbreviations

taut(t,n)	the boolean expression t contains no more than n variables and is a tautology (i.e. it yields the truth value 1 for all possible assignments of the truth values 1 and 0 to the variables of t)
true (t)	the boolean expression t does not contain any variable and the evaluation of t yields the truth value 1.
subst(t,n,x)...	the result of substituting the truth value x for the n-th variable in the boolean expression t.

A suitable program for the computer tree is:

```
1: if ¬S then go to 1;
   If n=0 then
          begin y:= true(t); U:= true; go to stop
          end;
```

```
if n≥1 then
     begin
     t':= subst(t,n,1);  n':= n-1;  V_1:= true;
     t":= subst(t,n,0);  n":= n-1;  V_2:= true;

  2: if ¬(T_1∧T_2) then go to 2;
     y:= y'∧ y";  U:= true; go to stop
     end;
stop: ;
```

Explanation: After having loaded and started this program by the starting routine (Section 2) all modules of the tree wait in the waiting cycle 1: if ¬ S then go to 1; with the exception of the top module whose sensor bit S has been set to true by step 5 of the starting routine.
Therefore the top module leaves the waiting cycle and inspects the data t (for example: $\neg(x_1 \wedge (\neg x_1 \vee x_2)) \vee x_2$) and n (for example: 2) loaded by step 3 of the starting routine. Normally, n will be ≠ 0. Therefore, the top module loads the storage regions t of its left and right sons with the boolean expressions subst(t,n,1) (in the example: $\neg(x_1 \wedge (\neg x_1 \vee 1)) \vee 1$) and subst(t,n,0) (in the example: $\neg(x_1 \wedge (\neg x_1 \vee 0)) \vee 0$), respectively and sets the variable n of both sons to n-1 (in the example: 1). Then it signalizes to its sons that they should start to work by setting their sensor bit S to true (by the instructions V_1:= true; V_2:= true, respectively).
Then it is trapped in the waiting cycle 2: if ¬ $(T_1 \wedge T_2)$ then go to 2; which can be left only if both sons will have announced the termination of their work by setting the sensor bits T_1 and T_2. If this happens the top module combines the results of both sons (stored in y' and y'', respectively) by y:=y'∧ y'' to yield its own output y and signalizes to its own father (which is the external world in case of the top module) that it has finished its work by setting U:= true;.
In the same way the two sons delegate the four subtasks (characterized, in our example , by the data

$\neg(1 \wedge (\neg 1 \vee 1)) \vee 1$, $\neg(0 \wedge (\neg 0 \vee 1)) \vee 1$, $\neg(1 \wedge (\neg 1 \vee 0)) \vee 0$,

$\neg(0 \wedge (\neg 0 \vee 0)) \vee 0$)

to four grandsons and so on.

Finally, at a level that is characterized by the parameter n=0, (in our examples, this is the level of the grandsons) the modules of that level will evaluate the boolean expressions by the instruction y:=true(t) and signalize the completion of their tasks to their respective fathers by the instruction U:= true;.

Their fathers will combine their results by $y:=y' \wedge y''$ and, again, signalize the completion of the task to their own fathers, and so on, until the top module is reached, again.

(In our example, the four modules at level 2 all compute the result 1, the two modules at level 1, then, combine these results producing, again, the result 1, and, finally, the top module combines these two results producing the answer 1. It signalizes termination by sending <u>true</u> via line 1 by the instruction U:= <u>true</u>).
Thus, by the execution of the above program on the computer tree, a computational "wave" is generated in the tree, which, first, distributes downward until the level n is reached and, then, contracts again upward.

<u>Remark 1:</u>
We observe that the time complexity of this computation on the computer-tree is $O(n)$ whereas it is $O(2^n)$ when the same recursive algorithm is executed on an ordinary computer.

This is so, because the computer tree allows a "material incarnation" of the procedure bodies when recursive procedure calls occur and, therefore, allows the parallel execution of logically independent procedure calls (as, for instance, the calls taut(subst(t,n,1),n-1) and taut (subst(t,n,0),n-1) in Example 1), whereas in an ordinary computer, by means of wellknown stack mechanisms or by transforming recursions into iterations the execution of recursive procedure calls has to be sequentialized.

This drastic collapse of time Complexity would make computer-trees much superior to ordinary Von-Neumann-computers. Of course, we must pay for this gain in time complexity by an equally drastic explosion of hardware complexity.

Therefore, one might argue, that computer-trees are of no practical value. We do not want to fix a final standpoint in this controversy.
Rather, before rejecting the idea of computer-trees, we would like to further explore its possibilities by giving some more examples.

<u>Example 2:</u>
A concrete realization of a computer tree can have only a fixed finite number of levels, say N levels.
How can we solve the tautology problem for arbitrary n on such a tree and how will the time complexity be influenced by this restriction ?

A suitable program would be:

```
if  ¬S  wait;                                                    +)
if  k=0  then
         begin  y:=true(t,n); U:=true; goto stop
         end;
if  k≥1  then
         begin
         t':=subst (t,n,1);
         t":=subst (t,n,o);
         n':=n":=n-1; k':=k":=k-1;
         $V_1$:=$V_2$:=true;
         if  ¬($T_1$ ∧ $T_2$)  wait;
         y:=y'∧ y"; U:=true; goto stop
         end;
stop: ;
```

Explanation:
k is an additional parameter that must be set to N at the beginning.
true(t,n), now, is a procedure that decides whether the boolean expression t contains no more than n variables and is a tautology. We suppose that this procedure works iteratively (by checking through the 2^n possible truth valuations) or is a recursive procedure sequentially executed in the respective module (this, again, needs $O(2^n)$ steps).
The time complexity of the above program on the computer tree, then, is $O(2^{n-N})+N$.
Hence,

$$\frac{\text{computation time on ordinary computer}}{\text{computation time on computer tree}} \simeq \frac{2^n}{2^{n-N}} = 2^N$$

This means, that in any case we have a constant speed-up of 2^N. If, for instance, $N \simeq 15$ (which seems to be realistic for present day hardware) then a constant speed up of 1ooo - 1oo.ooo is possible. This corresponds to what has been achieved by hardware improvement in one computer generation. Thus, we can say that though we cannot really convert $O(2^n)$ time complexity into O(n) we could perform a "one-generation-speed-up" by a new computer concept instead of new computer hardware .

+) if ¬S wait ; abbreviates α: if ¬S then goto α ;

Example 3:

Consider the merge-sort algorithm, see [13] :

$$\text{sort}(a) := \begin{cases} a, \text{ if length}(a)=1, \\ \text{merge}(\text{sort}(\text{left}(a)), \text{sort}(\text{right}(a))), \\ \text{if length}(a) > 1 \end{cases}$$

where we used the following abbreviations

sort(a)	the result of sorting the data sequence a
length(a)...	the number of items in the sequence a
left(a).....	the left half of the sequence a
right(a)....	the right half of the sequence a
merge(x,y)..	the result of merging the sorted sequences x and y

A suitable program for this algorithm on the computer tree is:

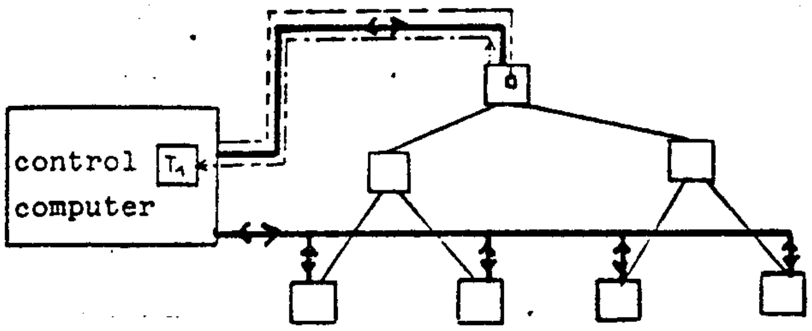
```
if  ¬S  wait;
if  length(a)=1  then
                 begin  y:=a; U:=true; goto stop
                 end;
if length(a) ≥1  then  begin
                 a':=left(a); V_1:=true;
                 a":=right(a); V_2:=true;
                 if ¬(T_1 ∧ T_2)  wait;
                 y:=merge(y',y"); U:=true; goto stop
                 end;
stop: ;
```

The time complexity of this algorithm on the computer tree is O(n) (where n:=length(a)) whereas it is O(n.log n) on an ordinary computer.

Example 4:

A new point of criticism against computer trees arises in Example 3: every module needs a lot of storage (more accurately the top module needs most, the modules at level 1 half of it, the modules at level 2 a quarter of it, and so on). However, we can avoid this by a more tricky program, which works according to the following idea: in every module provide just one storage "box" (for containing only one item). If this box is void (this is the case if the father has fetched the item from the box) compare the items contained in the boxes of the two sons and fetch the left or the right item depending on which one is greater.
We use this example to show how a computer tree can be connected with ordinary computers to yield useful algorithms.

For this we suppose that the modules of the lowest level and the top module in the tree can be accessed by an ordinary computer in the following way:



We change the mode of operation:

1. The control computer loads the input vector a into the boxes of the modules at the lowest level, one item in each box. Furthermore, it loads a special item "↓" into the boxes of all the other modules (↓ is defined to be greater than all the regular items).
2. Load the program P_2 into all modules of the tree with the exception of the bottom modules.
3. Load the program P_3 into the bottom modules.
4. Reset the sensor bits of all modules to <u>false</u>.
5. Simultaneously start all modules.
6. Execute control program P_1 in the control computer.

Program P_1:

```
       n:=1;
    1: if b'= ↑ then goto stop;
       if b'≠ ↓ then
                begin  y[n]:=b'; n:=n+1 end;
       V1:=true; if ¬T1 wait; V1:=false; if T1 wait;
       goto 1;
    stop: ;
```

Program P_2:

```
    1: if ¬S wait;
       if b',b"=↑ then
              begin b:= ↑ ;
                    U:=true; if S wait; U:=false;
                    goto 1
              end;
```

```
      if $b' \geq b''$ then
            begin $b := b'$;
                  $U :=$ true; if $S$ wait; $U :=$ false;
                  $V_1 :=$ true; if $\neg T_1$ wait; $V_1 :=$ false;
                  if $T_1$ wait;
                  goto 1
            end;
      if $b'' \geq b'$ then
            begin $b := b''$;
                  $U :=$ true; if $S$ wait; $U :=$ false;
                  $V_2 :=$ true; if $\neg T_2$ wait; $V_2 :=$ false;
                  if $T_2$ wait;
                  goto 1
            end;
```

Program P_3:

```
      1: if $\neg S$ wait;
         $b := \uparrow$;
         $U :=$ true; if $S$ wait; $U :=$ false;
         goto 1;
```

We hope that the above explanation of the basic idea is sufficient to understand the programs P_1, P_2, P_3. The variable b denotes the box. y will contain the resulting sorted vector. $\uparrow$ is a special item which is generated at the modules at the bottom to signalize the absence of regular items. $\uparrow$ is defined to be less than all the regular items.

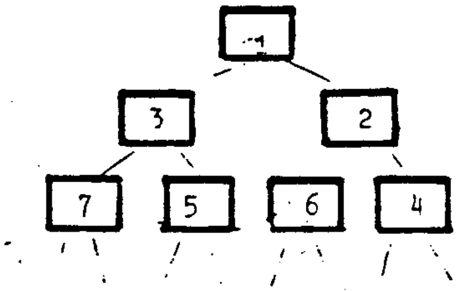
Note that the algorithm provides linear sorting with only one storage box needed in every module.

<u>Example 5:</u>

In this example, we want to show how a computer tree can be "abused" as a random access storage that may be arbitrarily extended without hardware changes by simply adding modules to the bottom of the tree. Simple load the following program to all modules except the bottom modules.

```
1: if ¬ S wait;
   if   k= 1 then
              begin  y:= storage [address];
                     U:= true; if S wait; U:= false;
                     go to 1
              end;
   if odd(k) then
              begin  k':= k/2; address':= address;          +)
                     V1:= true; if ¬ T1 wait; V1:= false;
                        if T1 wait;
                     y:= y';
                     U:= true; if S wait; U:= false;
                     go to 1
              end;
   if even(k) then
              begin k'':= k/2; address':= address;          +)
                     V2:= true; if  T2 wait; V2:= false;
                        if T2 wait;
                     y:= y'';
                     U:= true; if S wait; U:=false;
                     go to 1
              end;
```

Explanation: "k" is the number of the storage block (i.e. number of the module) in the tree and "address" is the address of the cell within the k-th storage block. We use the following enumeration of modules



The programs in the modules may be started by a control computer located at the top of the tree similar to Example 4, for instance, by the following intructions:

```
   k' := k; address' := address;
   V1 := true; if ¬ T1 wait;  V1 := false;  if T1 wait;
   y  := y';
```

Error messages may be generated at the bottom modules by the following program:

```
   1: if ¬ S wait;
```

+) integer division !

```
if k=1 then
        begin   y:= storage [address];
                U:= true; if S wait; U := false;
                go to 1;
        end;
if k>1 then     y:= "error";
                U:= true; if S wait; U:= false;
                go to 1;
        end;
```

Example 6:
We show how multiple recursions may be executed on computer trees. Consider, for instance, the following definition:

$$f(x) = \begin{cases} c, \text{ if } x=0 \\ h(x,g(x-1)), \text{ if } x>0 \end{cases}$$

$$g(x) = \begin{cases} d, \text{ if } x=0 \\ k(x,f(x-1)), \text{ if } x>0. \end{cases}$$

We extend our language by modifying the semantics of the go tc statement:

go to a;

should be interpreted by using the content of variable a as the label for the jump. Of course, this language feature is easily realized in machine languages. To be consistent in our high level example language we admit only integer labels from now on.

Then we may model the above double recursion by the following program (loaded to all modules in the tree) which again provides "material incarnations" of all procedure calls:

```
      if ¬ S wait;
      goto a;
1: if x=0 then
        begin y:= c; U:= true; goto 3
        end;
   if x≠0 then
        begin x':= x-1; a':= 2; V_1:= true;
              if ¬T_1 wait;
              y:= h(x,y'); U:= true; goto 3
        end;
2: if x=0 then
        begin y := d; U:= true; goto 3
        end;
```

```
if x≠ 0 then
        begin x' := x-1; a' := 1;  V_1 := true;
              if ¬ T_1 wait;
              y:= k(x,y'); U:= true; goto 3;
        end;

3: ;
```

Explanation: After initialization the program considers the label stored in a. Depending on whether it is 1 or 2 it executes the procedure body for the function f or g, respectively.

Example 7:
By using the goto-mechanism of Example 6 we can treat recursive procedures with more than two parallel procedure calls within one procedure body. Hence, the computer tree is a "universal" means for parallel execution of recursive procedures (though some modules are wasted by reducing a multiple branching to a binary one). Consider the following example:

$$f(x) = \begin{cases} g(x), \text{ if } p(x) \\ k(fh_1(x), fh_2(x), fh_3(x)), \text{ if } \neg p(x). \end{cases}$$

This can be transformed to the double recursion

$$(*) \qquad f(x) = \begin{cases} g(x), \quad \text{if } p(x) \\ k^*(fh_1(x), f^*(x)), \text{ if } \neg p(x). \end{cases}$$

$$f^*(x) = \tau(fh_1(x), fh_2(x)),$$

where $k^*(y,z) := k(y,z_1,z_2)$ and z_1,z_2 denote the first and second projection of z relative to the pairing function τ. Now,(*) may be programmed by means of the goto-mechanism of Example 6.

Example 8:
We now describe a program starting routine that operates as a typical tree program itself:

```
if ¬ S wait;
load (p);
p':=p'':=p;
V_1:=V_2:= true; if ¬(T_1∧T_2) wait;
          V_1:= V_2:= false;  if T_1∧T_2 wait;
U:= true; if S wait; U:= false;
goto begin;
```

Explanation: "load" is a routine transferring the storage region p to the private memory of the module, "begin" is the fixed initial address of programs in the private memory. In the bottom modules, a slightly different loading routine is necessary.

Conclusion:

We defined an unconventional computer structure called computer tree and a corresponding type of programming language. We presented a number of programming examples that may demonstrate the following aspects:

1. The computer tree theoretically allows for an execution of all types of recursive procedures via a "material incarnation" of procedure calls. This leads to a drastic collapse of time complexity in typical examples (see Examples 1, 3 and 7), which cannot be achieved even on array and vector computers.

2. The gain in time complexity must be purchased by an equivalent explosion of hardware complexity.

3. In any case, since computer trees with 2^{15} modules seem to be possible, a constant speed-up by a factor ≥ 1000 is possible for "hard" (i.e. exponential) problems. (see Example 2).

4. Furthermore, realizations of computer trees with only little storage per module seem to be of practical value for special applications. Computer trees of this type have the structure of a large storage with a processing element at every storage "box" (Example 4).

5. Computer trees may also be abused as extendible random access storages with unlimited extension capability (Example 5).

6. Computer trees should be considered as an additional hardware unit which may be profitably used in connection with a conventional computer and not as an exclusive alternative to conventional computers (Example 4).

Finally, we want to remark that what seems to be most important for the practical significance of computer trees is a hardware possibility that would allow flexibly connecting the modules at execution time. This would permit the execution of computations that need paths of different length in the tree. Some proposals for reconfigurable hardware have already been made,see,for instance, /9/ and /11/.

R e f e r e n c e s:

/1/ P.H. Enslow (ed.) Multiprocessors and Parallel Processing, J. Wiley, New York 1974.

/2/ T. Feng (ed.) Parallel Processing, Proc. of the Sagamore Computer Conference, August 2o-23, 1974, Lecture Notes in Computer Science, Vol. 24, Springer 1975.

/3/ J. Wilmink, M. Sami, R. Zaks (ed.) Microprocessing and Microprogramming, Proc. of the EUROMICRO Symposium, Oct. 12-14, 1976, Venice, North-Holland , 1977.

/4/ M. Feilmeier (ed.) Parallel Computers - Parallel Mathematics, Proc. of the IMACS(AICA)-GI Symposium, March 14-16, 1977, Munich, Munich, North-Holland 1977.

/5/ J.D. Nicoud, J. Wilmink, R. Zaks (ed.) Microcomputer Architectures, Proc. of the EUROMICRO Symposium Oct. 3-6, 1977, Amsterdam, North-Holland, 1977 (Preprints).

/6/ R. Albrecht Zur Struktur von Informationssystemen, in: J. Dörr, G. Hotz (ed.), Automatentheorie und formale Sprachen, Proc. of a Symposium, Oberwolfach, Oct. 1969, Bibliographisches Institut, Mannheim, 197o, pp. 493-5o5. 7

/7/ V.M. Glushkov, M.B. Ignatyev, V.A. Myasnikov, V.A. Torgashev Recursive Machines and Computing Technology, Proc. of the IFIP Congress 1974, North-Holland, 1974, pp. 65-7o. 2

/8/ F. Schwenkel Zur Theorie unendlicher Parallel-prozessoren, in: Proc. of the 4th GI-Jahrestagung Berlin, Oct. 9-12, 1974, Lecture Notes in Computer Science, Vol. 26, Springer, 1975, pp.355-366. 3

/9/ A. and K. Gostelow, A Computer Capable of Exchanging Processors for Time, in: Proc. of the IFIP Congress 1977, North-Holland, 1978, pp.849-853.

/1o/ W. Händler Aspects of Parallelism in Computer Architecture, in /4/ : pp.1-8. [illegible] 5

6 /11/ K.Vorgrimmler, P.Gemmar — Structural Programming of a Multiprocessor System, in: /4/: pp. 191-195.

/12/ B.Buchberger, J.Fegerl — Ein universeller Modul zur Hardware-Implementierung von Rekursionen, Laboratory Note, Univ.Linz, 1977.

/13/ A.V.Aho, J.E.Hopcroft, J.D.Ullman — The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.