数理解析研究所講究録
511巻 1984年 49-73

Evaluating graph representations with active nodes

沼尾正行　　志村正道
Masayuki NUMAO, Masamichi SHIMURA

東京工業大学工学部情報工学科
Department of Computer Science
Faculty of Engineering
Tokyo Institute of Technology
Oh-okayama, Meguro, Tokyo, 152

A method for evaluating a lambda expression is presented. In our system, a lambda expression is represented as a graph where nodes correspond to processes and arcs to channels. The graph evaluates itself as processes fork or terminate. Our method of evaluating lambda expressions is applicable to distributed systems of computer networks. Also our system has an ability to process structured data although it has no shared memories.

1.Introduction

As is well known, the functional programming languages are quite useful particularly in parallel machines since a value of an expression is independent of other expressions. Many methods to evaluate functional programming languages in parallel machines are proposed [Mago

1979, Keller, Lindstrom and Patil 1979, Darlington and Reeve 1981].

The programming methods which are useful in distributed systems are also proposed [Hoare 1978, Morrison 1978]. They have models which has many processes communicating one another. We propose a method to evaluate functional programs in such models. In our method, functional programs are expressed in graphs of processes which evaluate themselves. A method which evaluates the combinator graphs in such a manner was proposed [Kennaway and Sleep 1982]. Compared with other methods, our method has the following features.

(a) The programs need not be transformed into combinators. lambda expressions which involve variables and list structures are directly evaluated.

(b) Processes are never transferred in communication. Only data are exchanged. This feature cuts down communication cost.

2. Copy and reduction of graphs with active nodes

In our system, programs are stored in the form of graphs. Nodes of these graphs correspond to the processes and arcs correspond to communication channels. In the first evaluating step, the input reader transforms source texts into these graphs of processes. These processes communicate with one another and the graphs change their forms. The following two operations are primitives of graph transformation.

(a) The termination of processes of deleting nodes and connecting two of their neighbors.

(b) The forking of processes for the copy of nodes and arcs.

Figure 1 illustrates these two operations.

An example of reduction of graphs is shown in Figure 2. The car process sends a 'CAR' to a cons process and the processes terminate. The cons process connects the parent with the car-part of the children in order to choose the car-part of the list.

The definitions of functions are also stored in the form of graphs. More than one graphs may refer to a definition at the same time since operation (b) copies the definition.

After repeating these operations the reduced subgraphs output the results of the evaluations.

3. The list processing

The cons processes constitute list structures by building using the cons operators[Keller 1979]. They receive 'CAR' or 'CDR' that their parent nodes send. If they receive 'CAR' or 'CDR', they connect their parents to their car-parts or cdr-parts. This mechanism enables the 'CAR' or 'CDR' stream to select the parts of lists. The processes generate selectors 'CAR' or 'CDR' to manipulate list structures.

We introduce a language to describe processes. It is similar to Hoare's CSP(Communicating Sequential Processes) [Hoare 1978] except that it can describe procedure calls and recursions, and use ports and channels to communicate with other processes. Sources or destinations are specified by naming a port through which communications are to take place. The port names are local to the processes, and pairs of ports are to be connected by channels.

The description of the car process is

```
car_node::
   CHILD!SEL(CAR);
   stop(PARENT,CHILD).
```

A identifier followed by '::' is a process or procedure name.

The output command has the form

```
<port>!<expression>
```

The second line is a output command which outputs 'SEL(CAR)' to the port whose name is CHILD which is connected to the process at the child node. 'SEL(CAR)' is said 'structured expression'. Target variables of input commands which receive structured expression must have same structures. That is to say, a target variable which correspond to 'SEL(CAR)' must have the form 'SEL(<variable>)'.

The input command has the following form.

```
<port>?<target variable>
```

Inter-process communications use no buffers, in other words, corresponding input and output commands are executed simultaneously.

The statement

```
stop( <port1>, <port2> )
```

terminates the process, deletes the node and connects <port1> with <port2>.

The description of the cdr process is almost same as that of the car

process as

```
cdr_node::
   CHILD!SEL(CDR);
   stop(PARENT,CHILD).
```

where the second line is different from the car process. It does not send 'CAR' but 'CDR'.

The cons process is represented in the following description. The identifiers PARENT, CHILDCAR, CHILDCDR represent ports to the parent, the car-part and the cdr-part of this node.

```
consnode::  element,selector:var;
   [PARENT?SEL(selector) ->
      [selector=CAR -> stop(CHILDCAR,PARENT) ▯
       selector=CDR -> stop(CHILDCDR,PARENT) ] ▯
    PARENT?EOS() ->
      PARENT!'(';
      CHILDCAR!EOS(); CHILDCDR!EOS();
      o1ele(CHILDCAR);
      CHILDCDR?element;
      stop(CHILDCDR,PARENT) ].

o1ele(c) ::
   c?element; PARENT!element;
   [element='(' ->
       *[ element ≠ ')' -> o1ele(c) ]
       ; element := NIL ▯
    element ≠ '(' -> skip].
```

5.

The form

[<guard> -> <command list> ▯ <guard> -> <command list> ▯ ...]

is said an alternative command. An arbitrary one successfully executable guard is selected and the followed command list is executed. The expression which is put '*' before a alternative command is a repetitive command. It specifies as many iterations as possible of constitutent alternative command.

The cons process selects following three behaviors.

(a) If it receives 'CAR', it connects 'PARENT' with 'CHILDCAR' and terminates.

(b) If it receives 'CDR', it connects 'PARENT' with 'CHILDCDR' and terminates.

(c) Receiving the element EOS() indicates that the 'selector' stream is empty. In this case the cons process gets 'value' streams from 'CHILDCAR' and 'CHILDCDR', and outputs cons of these streams to the parent node.

If the list structure is given only EOS() as the selector stream, it outputs whole stream of the list sturucture's value. It is evaluated in a fully lazy manner which is compatible with the 'Lenient cons' [Friedman 1976]. The atom process which constitute leaves of list structures are described as

```
atom_node(atom)::
  *[PARENT?EOS() -> PARENT!atom].
```

where the argument 'atom' specifies the atom that the atom node outputs. To identify the tail of the list sturcture, the following nil process is used.

```
nil_node::
  *[PARENT?EOS() -> PARENT!'('; PARENT!')'].
```

It outputs a pair of parentheses.

The cons process also handle a 'color' element that argument selection nodes use. The argument selection node is explained in the section 4.

The car and cdr process do not fork, that is to say, definitions of functions are modified after evaluation at the first time. So, the graph of the function definition exhibits self-optimizing properties [Turner 1979] in spite of the parallel evaluation. Optimizations are suppressed at the variables whose values depend on the environments.

4. Processing function calls

The fun, graph and argument selector processes execute function calls. The form of process graphs for function calls are illustrated in Figure 3. The fun process is embedded in the calling function. It is connected to the definition of called function by a channel. The graph process ties up a input and a output of the function body.

Function calls are processed with the following steps.

(a) The fun and graph processes fork. This operation separates

the channel connecting these processes into two channels. One of them is used for a input channel and the other a output channel.

(b) Terminations of the fun and graph processes connect the input and output of the function body with those of the fun process. This operation replaces the fun process by the function body.

The graph after these operations is shown in Figure 4.

After executing above steps, every graphs which call this function share the function body. As the selector stream flows, constituent processes fork one by one from the root to the leaves. To command processes to fork, the signal 'FORK()' flows before the selector stream. The car and cons processes which handle 'FORK()' are detailed in the Appendix.

The input of the function body is connected to every fun nodes which call the function. The argument selector process selects the calling fun process out of them. For this purpose, the color generator process puts the 'color' element which indicates the calling graph in the selector stream. The argument is selected corresponding to the color.

The description of the fun process is

```
fun_node:: color:var;
   PARENT?FORK();
   PARENT?COLOR(color);
   [  fun_node
   ¦¦ FUNARG!FORK(); FUNARG!COLOR(color);
      FUNARG?GRAPH();
      [  stop(FUNARG,CHILD)
```

```
        || FUNARG!FORK(); stop(PARENT,FUNARG) ].
```

where 'FUNARG' is a port to the definition of the function.

The fork of a process is represented by

```
  [ <process> || <process> ].
```

which copies the node and the arc. The right process will use the channel that the process uses before. Other channels are connected to the left process. Variables of the original process is not shared but copied into each process.

A signal 'FORK()' and a color invoke a function call.

The graph process has three ports 'PARENT', 'VALUE' and 'ARG' which are marked in Figure 3. Corressponding with the fun process, we get the following definition.

```
graph_node:: selector,color,oldcolors:var; graph1(0).

graph1(color:var)::
   PARENT?FORK(); PARENT?COLOR(oldcolors);
   PARENT!GRAPH();
   [  graph1(color+1)
   || ARG!COLORREG(color);
      PARENT?FORK();
      [stop(PARENT,ARG)
      || VALUE!FORK(); VALUE!COLOR((color,oldcolors));
         stop(VALUE,PARENT)]].
```

The graph_node registers a color to the argument selector and put the

signal 'FORK()' and the color before the selector stream. Each color corresponds with the calling graph. Using this color, the argument selector can select the calling graph. The description of the argument selector is

```
argument_selector_node:: color,oldcolors,element:var;
                         asel1.

asel1::
  *[TO_GRAPH?COLORREG(color) -> stream(TO_GRAPH, color) ▯
    TO_BODY?COLOR((color,oldcolors)) ->
      [asel1 ¦¦ color!COLOR(oldcolors); stop(color,TO_BODY)]].
```

where 'TO_BODY' is a port of a channel to the function body and 'TO_GRAPH' the graph node. The statement

```
stream( <port>, <new port> )
```

creates a port <new port> and its channel which is same as <port>'s. This statement copies the channel.

To register a color to the argument selector, the graph_node sends COLORREG(<color>) and the argument selector creates the new port whose name is <color>.

5. Evaluating variables

To get a value of a variable, the place of the actual parameter which corresponds to the variable must be found. We introduce name elements

into selector streams to find the value. Variables in source text are transformed into name nodes in the graph. The name node sends a name to the lambda node and the lambda node selects a actual parameter.

The description of the name node is

```
name_node(name)::
    CHILD!NAME(name);
    stop(PARENT,CHILD).
```

which is similar to the car and cdr nodes except that it sends a name instead of a selector.

The lambda node tries to find a name in its name table that has names of formal parameters. If the name is found, the lambda node generates a selector stream to access the actual parameter. If not, it sends the name to the lambda node which corresponds to the outer lambda expression.

```
lambda_node(name_table):: name,element,color:var;
                          q:queue;
    PARENT?NAME(name);
    [  lambda_node(name_table)
    || q:=search_table(name,name_table);
       [not(empty(q)) -> CHILD!SEL(CAR);
                         *[CHILD!SEL(remove(q)) -> skip] []
        empty(q)      -> CHILD!SEL(CDR);
                         CHILD!NAME(name)];
       stop(PARENT,CHILD)].
```

The function 'search_table' searches a name table and the queue 'q'

is assigned a sequence of selectors to access the corresponding formal parameter. The predicate 'empty' checks whether the queue is empty or not. The function 'remove' removes a element from the queue. It fails if the queue is empty.

Strictly speaking, the lambda node needs to handle the signal 'FORK()' and the color. The description is elaborated like the car node in the Appendix.

The rules of transformation of the source text into the graph is shown in Figure 5. The lambda node can refer to the outer lambda node since the graph corresponding to the lambda expression replaces the fun node whose children involve the outer lambda node. Figure 6 illustrates an example of the graph which calculates Fib defined as

```
Fib         = ( 1, 1 ¦ mapcar(plus, Fib1(Fib)))
Fib1(x)     = ( (car(x),cadr(x)) ¦ Fib1(cdr(x)) )
mapcar(f,l) = ( f(car(l)) ¦ mapcar(f,cdr(l))).
```

which outputs the stream of the Fibonacci sequence endlessly. The print node at the top of the graph invokes a evaluation.

```
PRINT::
  CHILD!FORK();
  CHILD!COLOR(0);
  CHILD!EOS();
  *[CHILD?element -> print(element)].
```

The FUNARG problem is easily avoided in our evaluation method. You can easily get other environments connecting the channels to other lambda nodes. Our method has the same flexibility in environments as

the association list. But our method searches in a parallel manner. Self-optimizing properties that we discuss in the section 7 delete variables from global definitions of functions as graphs are evaluated. The variables are evaluated at the first time only.

6. Numerical operations and stream processing

Numerical function 'plus', for example, is described as follows.

```
plus_node:: selector,arg1,arg2,color:var;
   PARENT?FORK(); PARENT?COLOR(color);
   [plus_node
   || PARENT?EOS();
      stream(CHILD,CHILD1);
      CHILD1!FORK(); CHILD1!COLOR(color);
      CHILD1!SEL(CAR);
      CHILD1!EOS();
      CHILD!FORK(); CHILD!COLOR(color);
      CHILD!SEL(CDR); CHILD!SEL(CAR);
      CHILD!EOS();
      CHILD1?arg1; CHILD?arg2;
      PARENT!(arg1+arg2) ].
```

This node gets numerical values from their children and outputs a sum of them to the parent. Similarly, stream processing functions[Burge 1975] is described as

```
stream_processing:: color:var;
```

```
PARENT?FORK(); PARENT?COLOR(color);
[  stream_processing
|| PARENT?EOS();
   CHILD!FORK; CHILD!COLOR(color);
   CHILD!EOS();
   <the description of the stream processing> ].
```

which works like UNIX's filters[Richie 1974].

7. Self-optimizing properties

Generators of selector streams, the car, cdr and name processes do not fork. So, their nodes in definitions are always deleted and definitions are optimized. A subgraph which involves variables must not be optimized since their values do not fix. To avoid optimizations of variables, the argument selectors block selector streams until the car and cdr processes fork, that is, the argument selectors receive colors. The car and cdr processes fork when they receive colors as is detailed in Appendix.

If the optimization is restricted only in a function, the effect of optimization is small. We introduce global functions and variables for more optimizations. The global function does not change its definition during the evaluation. It can be replaced by the function bodies at the first evaluation.

Almost all functions except local defined functions are global defined functions. The global variable does not change its value during the evaluation. It is a constant variable in other word.

The global must be declared before evaluation. The input reader

transforms a occurence of a global function into the global_fun_node instead of the usual fun_node.

The global_fun_node does not fork before the function call. So, the fun_node of not only copied graph but also the definition is replaced by the called function body. This will fix the definition of the function. The body of the called function will melt in the calling function because of Self-optimizing properties. The global fun node enables the graph to be applied more self-optimizing properties.

The lisp compiler or the translater to combinators deletes variables and improve the execution speed. Self-optimizing properties in our system has same effects as the compilation.

8. Sketches for the implementation

In this section, we consider how to implement our system. Since processes do not use shared memories, our method is useful in distributed systems.

Consider the square array of computers connected by packet communication lines. It is illustrated in Figure 7. All computers can communicate with four neighbors. Processes which constitute a graph can be assigned to these computers since all communications are restricted to channels. Since each computer contains definitions of functions, these functions are executed in a arbitrary computer. The process for load balancing is inserted between the name processes and the top-level lambda processes. This process chooses a computer which has the least load when new functions are called. All computers can access any resources distributed in the system by calling the resource handling functions.

6.

Graphs are represented by linked lists in each computer. Nodes of list represent process images. Channels to other computers are relayed by communication driver processes which controls packet communication lines. Channels are not created dynamically but only copied by the fork of processes. Their connections are identified by channel numbers in the packet communication line. So, the system needs no consistent identifiers of process beyond each computer.

Different machines can be mixed in our system. For example, consider the list structure machines which execute only the cons processes. From the point of other machines, this machine behaves as if it had process graphs, but usual binary pointer structures represent these processes. Since the usual binary pointer structures require less memory space than those constructed by cons processes, the list structure machines make it easy to construct large data bases.

In this paper, all processes are described by the CSP-like language. It is easy to transform these descriptions into a conventional deterministic sequential programs introducing statements 'fork', 'stream', 'communicate' and 'stop'. The statement 'communicate' has to wait more than one port exchanging data for absorbing nondeterminism in CSP.

9. Conclusion

We propose a method to evaluate lambda expressions with processes which communicate one another. Although processes are distributed on graphs, our system can directly evaluate expressions involving variables and list structures. Our system also has some self-optimizing properties.

References

Bernstein,A.J. : "Output guards and nondeterminism in 'Communicating Sequential Processes,'" ACM TOPLUS, Vol.2, No.2, pp.234-238(1980).

Burge,W.H. : "Stream processing functions," IBM J. Res. Develop., Vol.19, No.1, pp.12-25(1975).

Darlington,J. and M.Reeve : "ALICE:A multi-processor reduction machine for the parallel evaluation of applicative languages," Proc. Functional Programming Languages and Computer Architecture, pp.65-75 (1981).

Hoare,C.A.R. : "Communicating Sequential Processes," CACM, Vol.21, No.8, pp.666-677(1978).

Keller,R.M., G.Lindstrom and S.Patil : "A loosely-coupled applicative multi-processing system," Proc. AFIPS Conf. NCC, pp.613-622(1979).

Kennaway,J.R. and M.R.Sleep : "Expressions as processes," Record of the 1982 ACM Conference on LISP and Functional Programming, pp.21-28 (1982).

Friedman,D.P., and D.S.Wise : "CONS should not evaluate its arguments," in Michaelson and Milner(eds.), Automata, Languages and Programming, Edinburgh University Press, pp.257-284(1976).

Mago,G.A. : "A network of microprocessors to execute reduction

languages," Two parts, International Journal of Computer and Information Sciences, Vol.8, No.5, pp.349-385 and Vol.8, No.6, pp.435-471(1979).

Morrison,J.P. : "Data stream linkage mechanism", IBM Systems Journal, Vol.17, No.4, pp.383-408(1978).

Ritchie,D.M. and K.Thompson : "The UNIX time-sharing system," CACM, Vol.17, No.7, pp.365-375(1974).

Turner,D.A. : "A new implementation technique for applicative languages," Software-practice and experience, vol.9, pp.31-49(1979).

Appendix

```
car_node:: color:var;
   [PARENT?FORK() -> CHILD!FORK() ▯
    PARENT?COLOR(color) ->
       [car_node ¦¦ CHILD!COLOR(color); car_node] ▯
       CHILD!SEL(CAR) -> skip ]
   stop(PARENT,CHILD).
```

In this description the signal 'FORK()' and the color can pass the car node even if a selector is waiting to output. If the color is passed, the car node forks. To implement these features, this description has "output guards"[Bernstein 1980].

```
consnode:: element,selector:var; q:queue;
   PARENT?element; cons1(element).
```

```
cons1(element:var)::
   *[FORK():=element -> addqueue(q,element);
                        PARENT?element;
                        [consnode ¦¦ cons1(element) ] ▯
     SEL(selector):=element
      -> [selector=CAR
            ->*[element:=remove(q) -> CHILDCAR!element];
              stop(CHILDCAR,PARENT)                    ▯
          selector=CDR
            ->*[element:=remove(q) -> CHILDCDR!element];
              stop(CHILDCDR,PARENT)                    ] ▯
     color=undefined; COLOR(color):=element ->
        addqueue(q,element)                              ▯
     EOS():=element
      -> PARENT!'(';
         addqueue(q,EOS());
         *[element:=remove(q) -> CHILDCAR!element;
                                 CHILDCDR!element];
         o1ele(CHILDCAR);
         CHILDCDR?element;
         [element='(' -> stop(CHILDCDR,PARENT) ▯
          element≠'(' -> PARENT!'¦';
                         PARENT!element;
                         PARENT!')'          ]].
```

The statement 'addqueue' adds a element to a queue and 'remove' removes an element from a queue. It fails if the queue is empty. The initial value of the queue is empty.

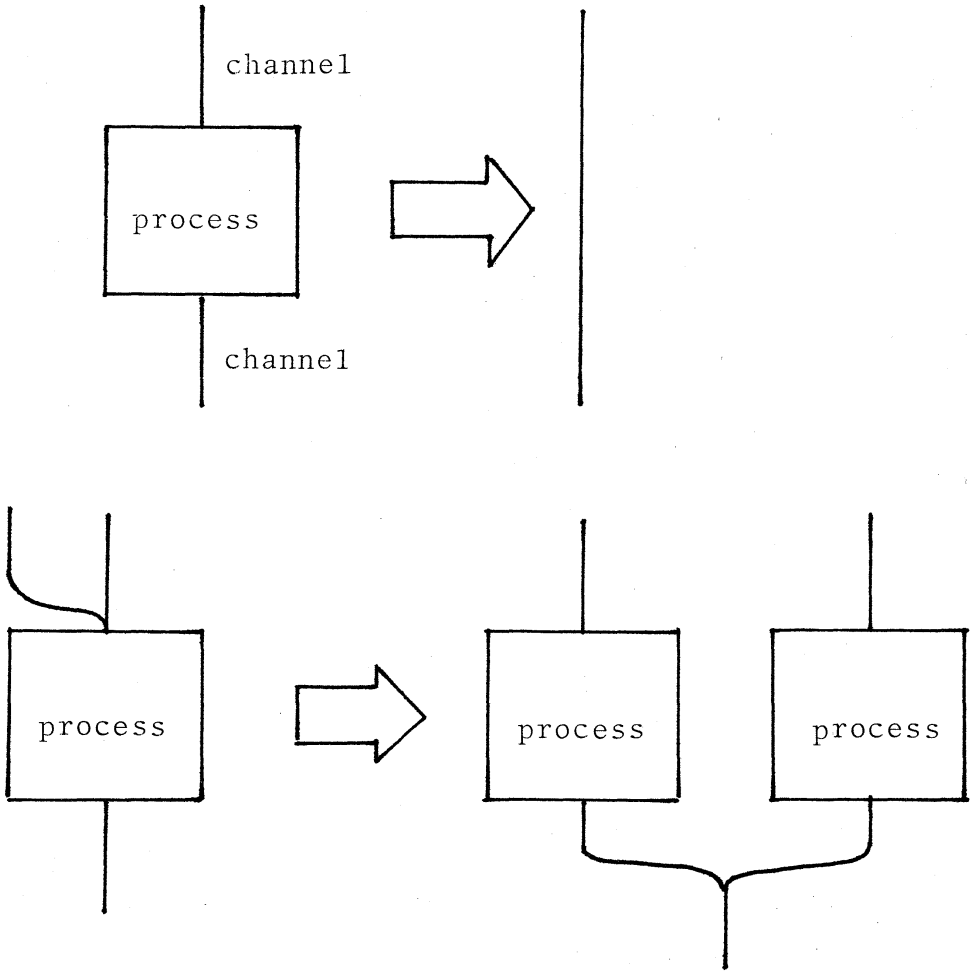


Figure 1. The termination and fork of processes

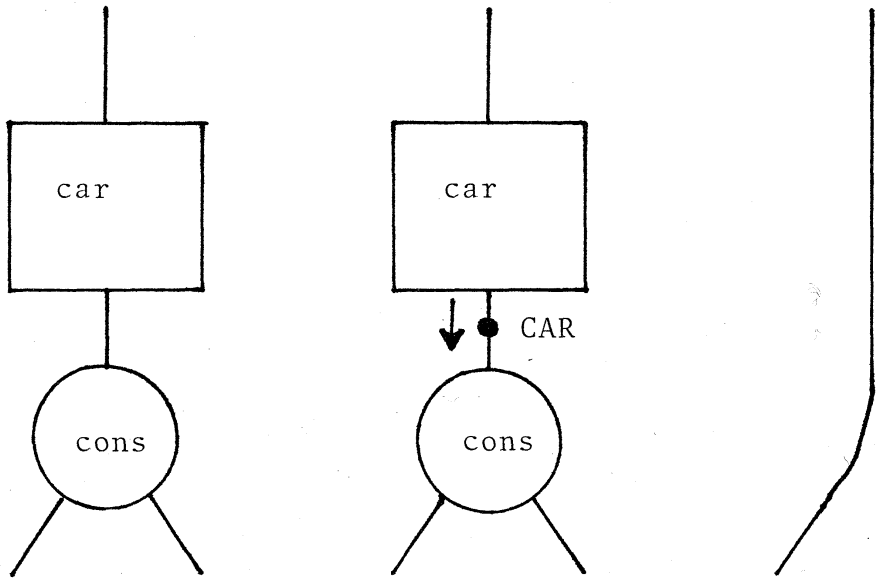
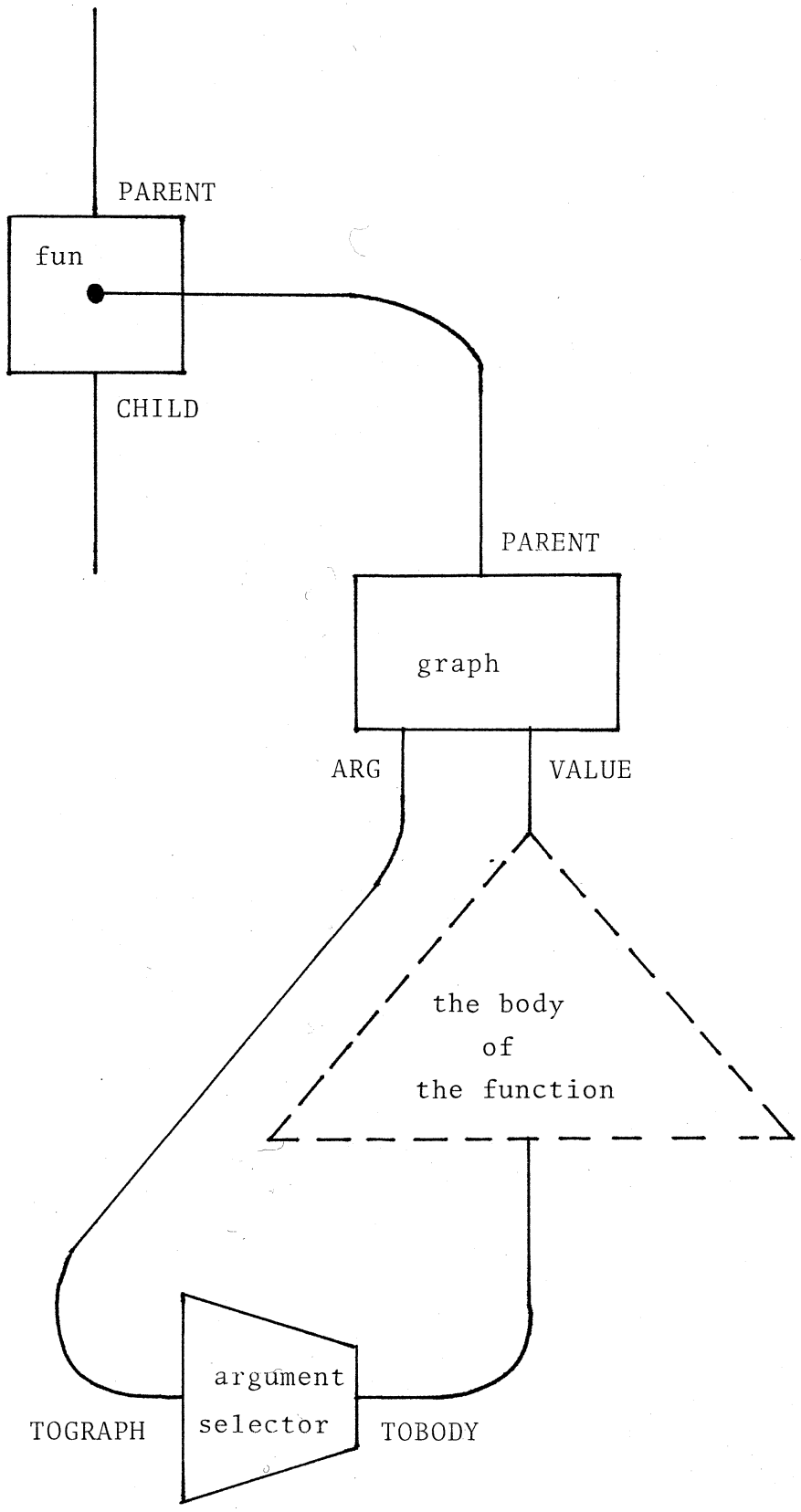


Figure 2. An example of the reduction



Fugure 3. The form of process graphs for function calls

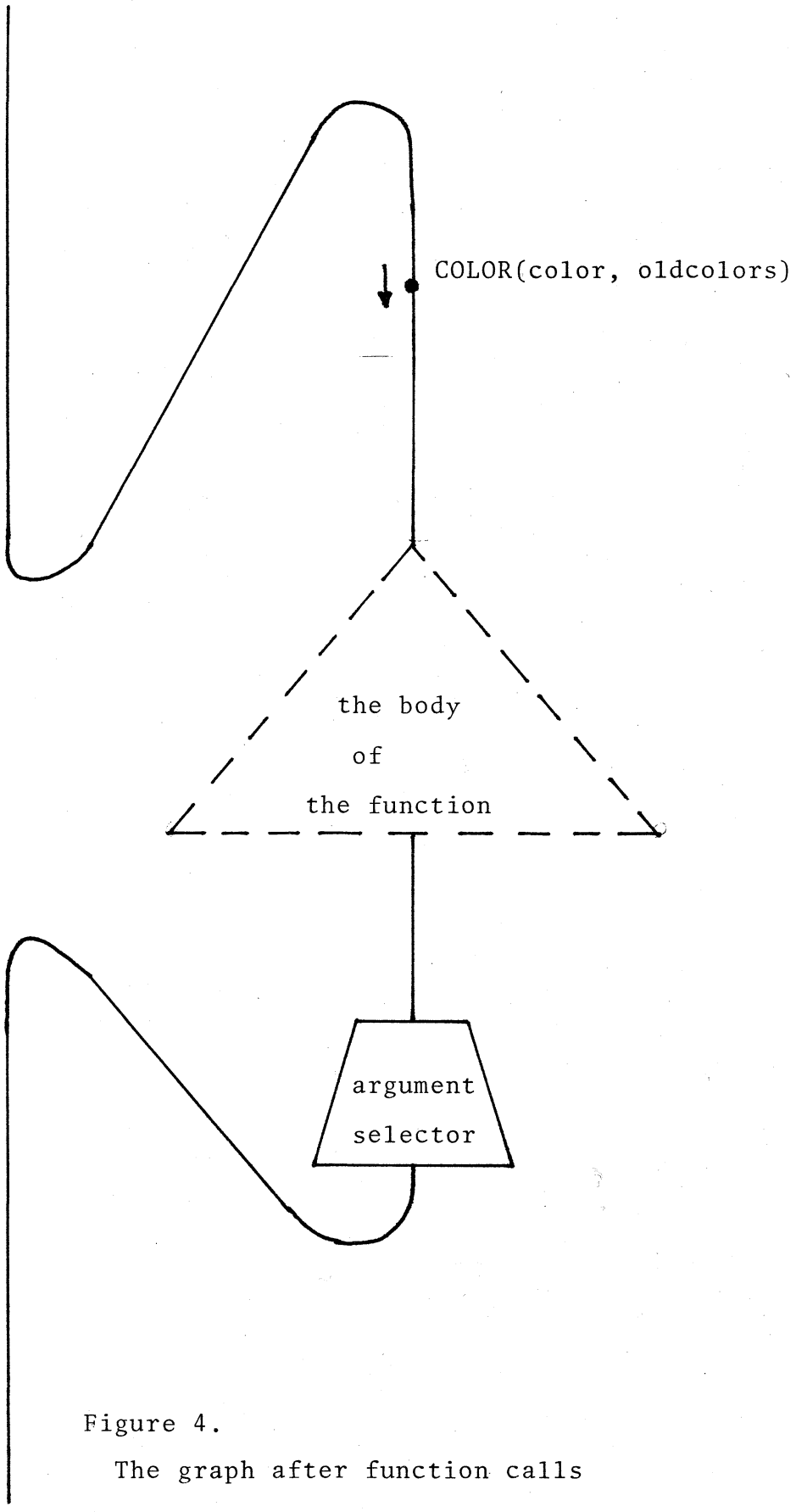
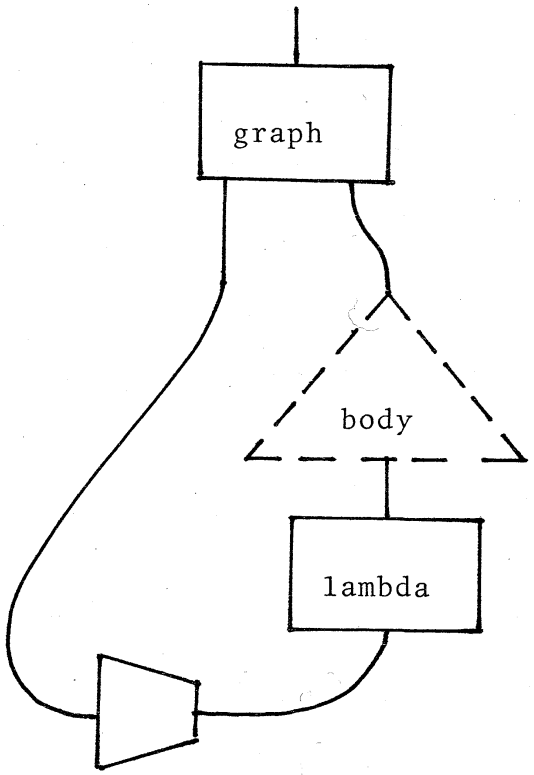
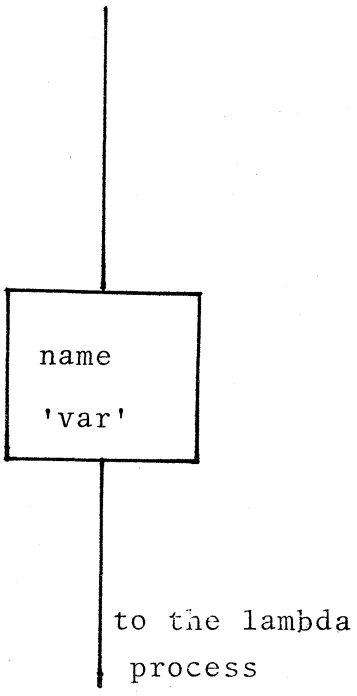


Figure 4.

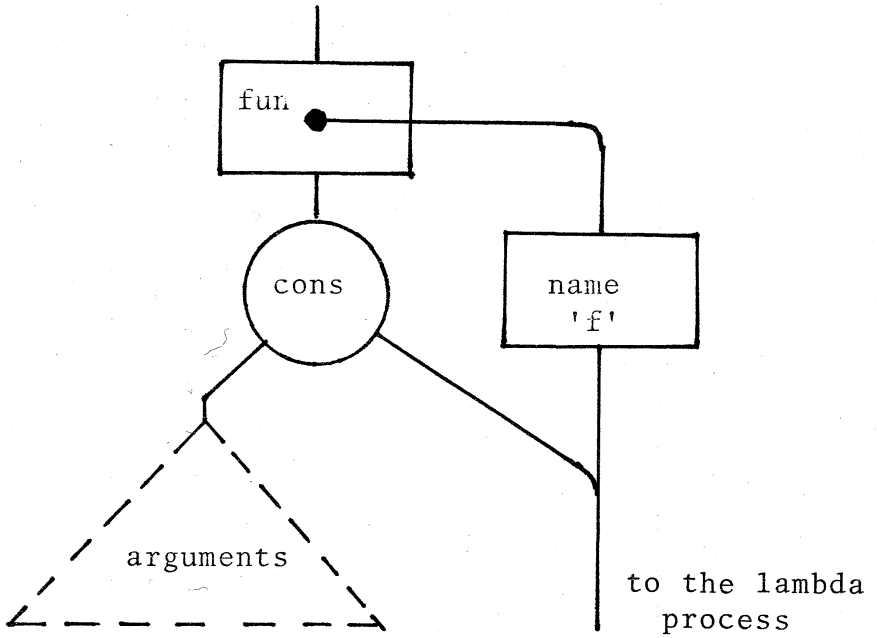
The graph after function calls



(a) lambda(⟨argument list⟩ , ⟨body⟩)



(b) ⟨var⟩



(c) f(⟨arguments⟩)

Figure 5. The rules of transformation of the source text into the graph

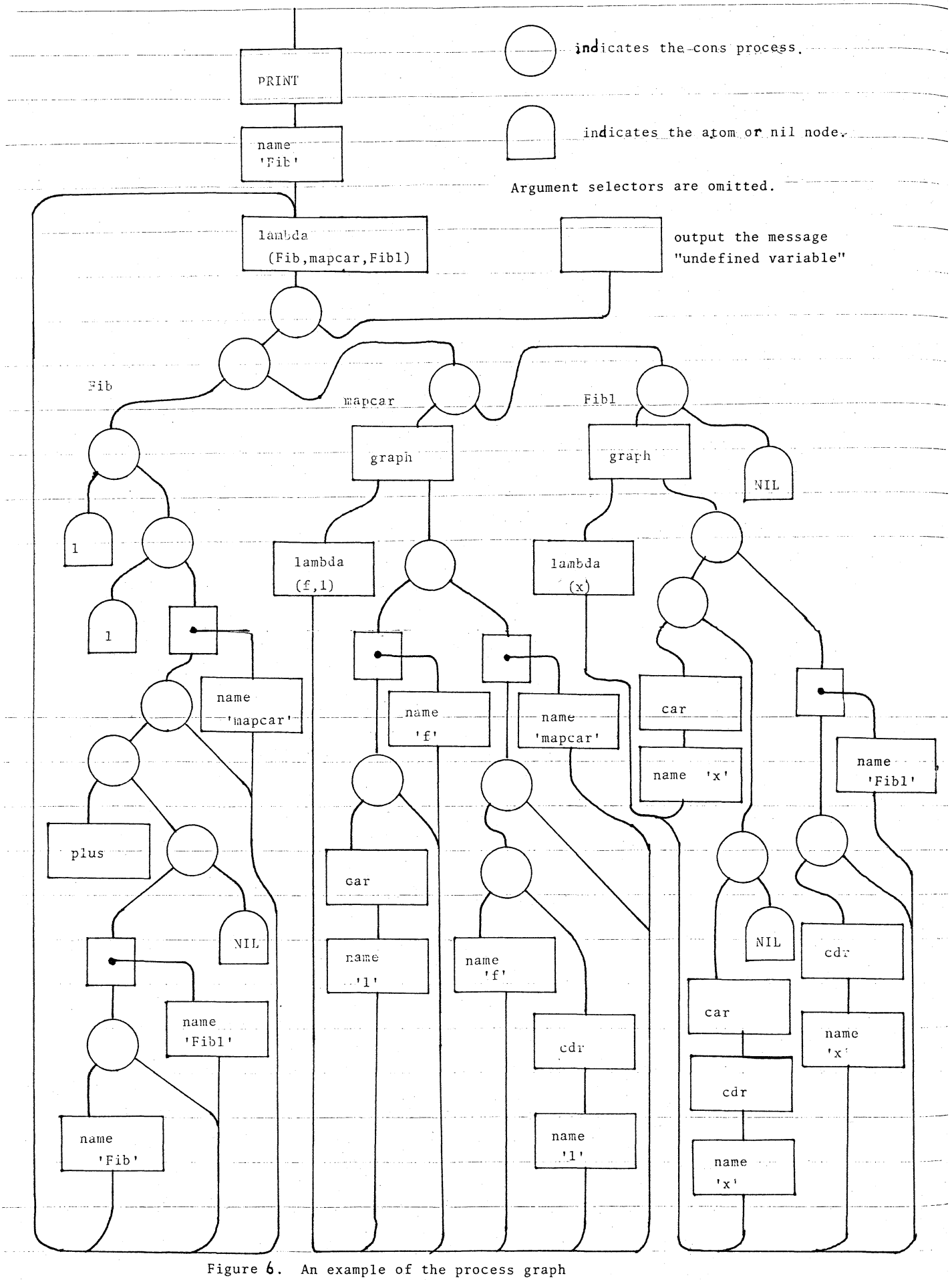


Figure 6. An example of the process graph

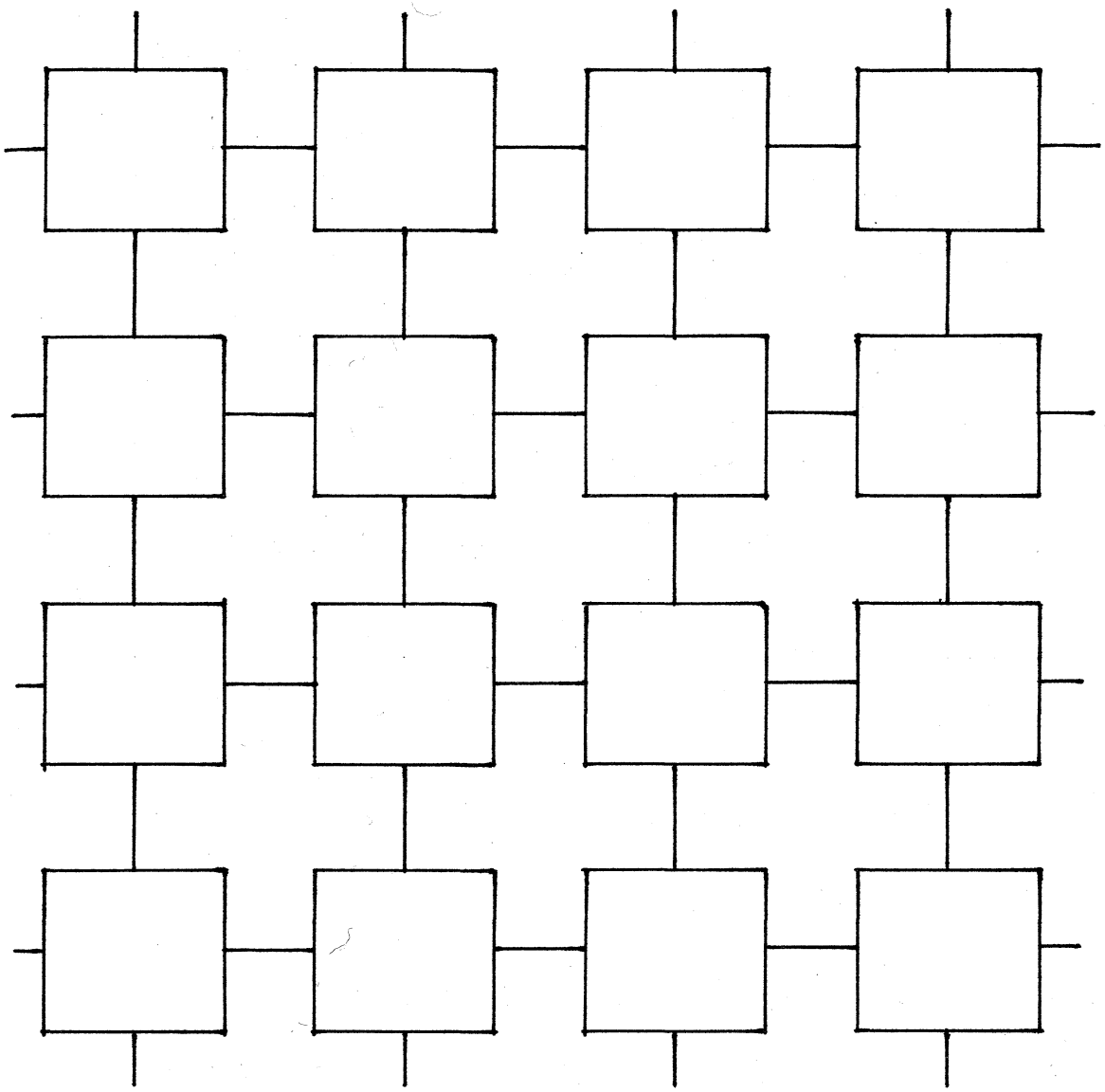


Figure 7. The square array of computers