TRYK: RECAU (06) 12 83 55

PB - 198 Jensen & Schmidt: Pascal Semantics

ISSN 0105-8517

Pascal Semantics by a Combination of Denotational Semantics and High-level Petri Nets

Kurt Jensen
Erik Meineche Schmidt

DAIMI PB - 198
January 1986

This paper will appear in "Advances of Petri Nets 1986" in Lecture Notes of Computer Science, Springer Verlag.

PASCAL SEMANTICS BY A COMBINATION OF DENOTATIONAL SEMANTICS AND HIGH-LEVEL PETRI NETS

Kurt Jensen and Erik Meineche Schmidt
Computer Science Department
Aarhus University, Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

This paper describes the formal semantics of a subset of PASCAL, by means of a semantic model based on a combination of denotational semantics and high-level Petri nets. It is our intention that the paper can be used as part of the written material for an introductory course in computer science.

Contents

1. INTRODUCTION

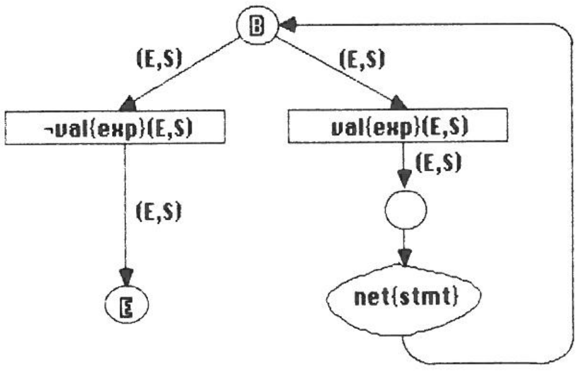
This paper describes the semantics of a subset of PASCAL. Programs are translated into high-level Petri nets, where the token-colours describe environments and stores. The reader is assumed to be familiar with the basic ideas behind high-level Petri nets. If not, one of the following papers should be consulted: [1, 4, 5, 6].

The translation is syntax-directed, in the sense that each kind of declaration or statement is mapped into a high-level Petri net. When a statement is aggregated from several other statements, the net is built in the usual algebraic way by combining the subnets of the constituent statements. As an example, the net of an if statement is built from the two subnets describing the then-part and the else-part.

More formally we shall define a (recursive) function, net{...}, mapping well-defined program-parts into high-level Petri nets. As an example we represent a while-statement of the form

while exp do stmt

by the following net



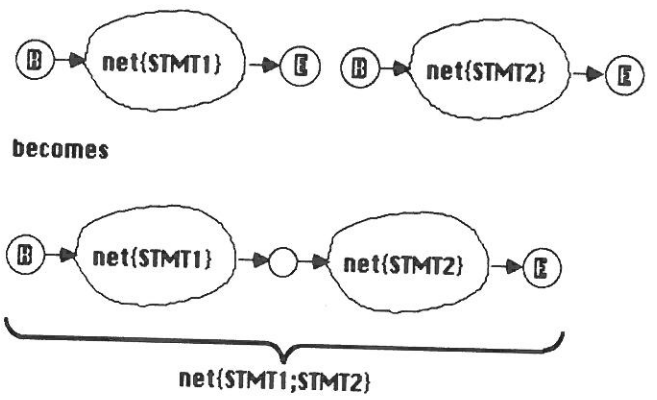
This example illustrates a number of properties, which apply to all nets obtained by the function net{...}.

First of all we observe, from the arc-inscriptions, that each place has token-colours, which are pairs. The first component, normally denoted by E, is an environment describing the current binding of names to their denotations (the name of a simple variable is bound to an address (location)). The second component, normally denoted by S, is a store describing the current binding between addresses and their contents. Environment and store are standard concepts from denotational semantics, and they will be defined and explained more carefully in section 2.

Secondly, we shall, in our arc-inscriptions and transition-inscriptions, use functions, such as val{exp}(E,S), to examine and update the environment and store. This kind of notation is also borrowed from denotational semantics and we shall return to it in section 2.

Thirdly, we observe that each net{...} has two distinguished places, indicated by an inscribed B and E (for begin and end). When a statement (or declaration) is ready for execution, its B-place is marked and the token-colour describes the current environment and store When execution of a statement finishes, the E-place is marked and the token-colour describes the new environment and store obtained by the statement.

Sequential execution of two statements (or declarations) is obtained by "gluing" together the E-place of the first statement with the B-place of the second:



To make this work, in a correct way, we require that E-places never have outgoing arcs (before composition with the B-place of the succeeding statement/declaration). Without this restriction there could be a choice, whether to resume execution of a statement or continue to the next.

We define the semantics of an arbitrary PASCAL program, PROG (of the subset considered), by the high-level Petri net, net{PROG}. Initially, this net has only a single token, which is positioned at the B-place. Since a PASCAL program behaves in a deterministic way when input is fixed, the net has only a single possible firing sequence.†) During this firing sequence (representing the execution of the program) the token moves from the B-place through the net in order to reach the E-place. By its position the token represents the current progress of execution (the program counter), while the colour represents the current environment and store. In addition to this (E,S)-token, the net may have other "auxiliary" tokens describing different items, like parameter-values, environments to be preserved for later use, number of remaining rounds in a for statement, etc.

---

†) The treatment of I/O is explained in section 8.

Having described the main ideas behind our semantic model, we now give a brief description of the history of our approach together with the purpose we want to achieve by it.

Our work with this type of high-level Petri nets as a semantic tool started around 1980. The first author was involved in the definition of a Concurrent Pascal like language [8] and a system description language, Epsilon [7]. The second author used the nets for teaching semantics of programming languages to first year computer science students, and the material presented in the current paper builds heavily on this work, although the net-notation has been rather heavily modified. Recently a similar approach has been published for CSP in [2].

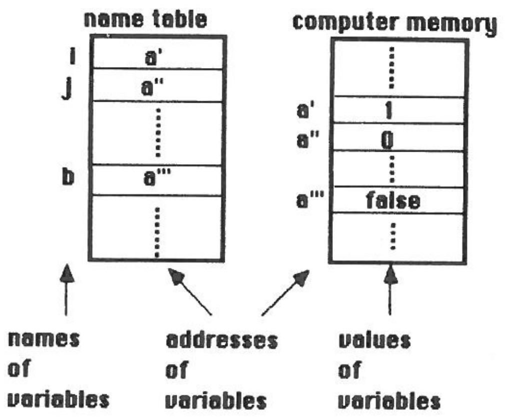
The main purpose of the approach is to use the semantic model as a didactic tool in the teaching of programming languages. It has never been the intention that programmers should verify the correctness of even small or medium-size programs, by manually constructing the appropriate high-level Petri nets. Instead the semantics is intended to be an intuitive, easy-to-understand and yet precise description of fundamental constructs in programming languages. The extent to which this goal has been achieved will be discussed in the concluding chapter.

The remaining part of the paper is organized as follows. Sections 2-8 contain our semantic definition of a non-trivial part of the PASCAL language. The reader is assumed to be familiar with the basic concepts of PASCAL, and we shall use the terminology introduced in [3]. Section 9 gives three examples showing how nets are constructed for small PASCAL programs. Section 10 contains a number of exercises. Section 11 is the conclusion. It discusses the adequateness of the semantic model, and how it is possible to extend the semantics to cover the remaining parts of PASCAL. It also describes the experience, over the last 5 years, with the use of this teaching material in the introductory computer science course at Aarhus University.

2. ENVIRONMENT AND STORE

When a PASCAL program is executed on a computer, the memory contains a word for each (simple) variable in the program. Each word is identified by an address by means of which we can examine or update the current contents of the word. As an example we show what the memory may look like, immediately after execution of the statement b:=false, in the following simple program.

```
PROGRAM P(INPUT,OUTPUT);
  VAR
    I,J: INTEGER;
    B:   BOOLEAN;
  BEGIN
    I:=1; J:=0;
    B:=FALSE;
      .
      .
      .
      .
  END (*P*).
```



In the semantics of PASCAL we shall represent the name table and the memory by two partial functions, E and S, known as <u>environment</u> and <u>store</u>.

$$\begin{tikzcd} \text{names of variables} \arrow[r, "E"] & \text{addresses of variables} \arrow[r, "S"] & \text{values of variables} \end{tikzcd}$$

In the example above we have

E(i)	=	a'	S(a')	=	1
E(j)	=	a"	S(a")	=	0
E(b)	=	a'''	S(a''')	=	false

The functions E and S will also be known as <u>catalogues</u>, and to manipulate them we introduce the following notations where C is an arbitrary catalogue.

$\overline{C}$ is the domain of C, i.e.

$$\overline{C} = \{x \mid \exists y:\ C(x) = y\}$$

$C[x\leftarrow y]$ is the catalogue which is obtained from C by relating x to y, i.e.

$$C[x\leftarrow y](z) = \begin{cases} C(z) & \text{if} \quad z \in \overline{C}-\{x\} \\ y & \text{if} \quad z = x \end{cases}$$

and $\overline{C[x\leftarrow y]} = \overline{C} \cup \{x\}$

and as shorthands (C' is an arbitrary catalogue and Λ is the empty catalogue $(\overline{\Lambda} = \emptyset)$):

$$C[x_1,x_2,\dots,x_n \leftarrow y_1,y_2,\dots,y_n] = C[x_1 \leftarrow y_1][x_2 \leftarrow y_2]\dots[x_n \leftarrow y_n]$$
$$[x_1,x_2,\dots,x_n \leftarrow y_1,y_2,\dots,y_n] = \Lambda[x_1,x_2,\dots,x_n \leftarrow y_1,y_2,\dots,y_n]$$
$$C[C'] = C[x_1,x_2,\dots,x_n \leftarrow y_1,y_2,\dots,y_n] \quad \text{where } [x_1,x_2,\dots,x_n \leftarrow y_1,y_2,\dots,y_n] = C'$$

$$[\ ] = \Lambda$$

To make the PASCAL semantics more readable we shall use the notation

ref{v}(E,S) and val{v}(E,S)

to denote, respectively, the address and the value of a variable v, calculated with respect to the environment E and the store S. The special brackets {...} surrounding v indicate that v is part of the program text.

From the example above, it is easy to see that the following is a reasonable definition of the functions ref and val

$$\text{ref}\{v\}(E,S) = E(v)$$
$$\text{val}\{v\}(E,S) = S(E(v))$$

We will also have to calculate values for more complicated expressions, such as $(x+y-1)*z$ or $x \neq y$. To do this we generalize the definition of val in the following way:

$$\text{val}\{c\}(E,S) = c$$
$$\text{val}\{exp_1 \oplus exp_2\}(E,S) = \text{val}\{exp_1\}(E,S) \oplus \text{val}\{exp_2\}(E,S)$$

where c is an arbitrary constant and $\oplus$ an arbitrary (dyadic) operator applicable to the expressions exp_1 and exp_2. As an example we get

$$\text{val}\{(x+y-1)*z\}(E,S) = (\text{val}\{x\}(E,S)+\text{val}\{y\}(E,S)-1)*\text{val}\{z\}(E,S)$$
$$\text{val}\{x \neq y\}(E,S) = \text{val}\{x\}(E,S) \neq \text{val}\{y\}(E,S)$$

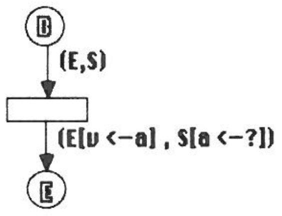
The definition of ref and val will be extended in later chapters when we include dynamic variables and record variables. Then it will also become apparent that in general ref depends on both E and S, and environments can bind names to other kinds of objects than just addresses.

3. VARIABLE DECLARATIONS, ASSIGNMENT STATEMENTS, BLOCKS AND PROGRAMS

A variable declaration of the form

```
          v : t
```

where t is a simple type (i.e. boolean, character, integer, real or pointer) is represented by the following net



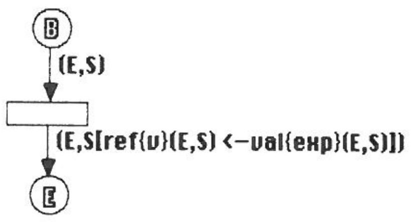
where a is a new address not appearing in S.

The variable name v is bound to an unused address which in turn is bound to the special value ?, representing the fact that the initial value of v is undefined. It should be obvious how to extend the semantics above to the case where more variables, with the same type, are declared simultaneously.

An assignment statement of the form

```
          v := exp
```

is represented by the following net

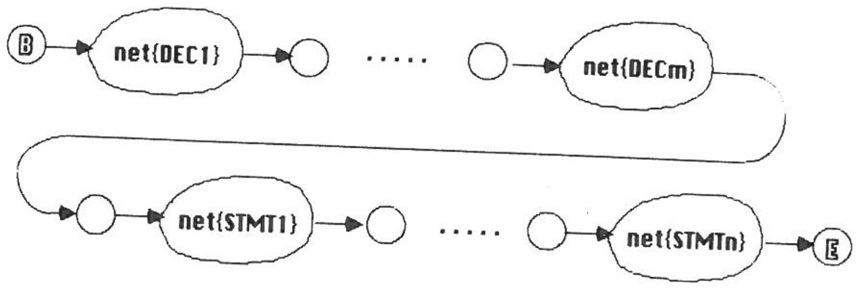


The expression exp is evaluated, and its value is associated with the address referenced by the variable v.

A block of the form

```
          var
             DEC1;...;DECm;
          begin                        (m≧0)
             STMT1;...;STMTn           (n≧1)
          end
```

is represented by the following net



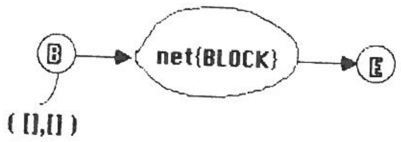
The net is obtained by "gluing" together the subnets which represent the constituent parts of the block.

A <u>program</u> of the form

<u>program</u> name(...);
    BLOCK

is represented by the following net

net{PROG}:



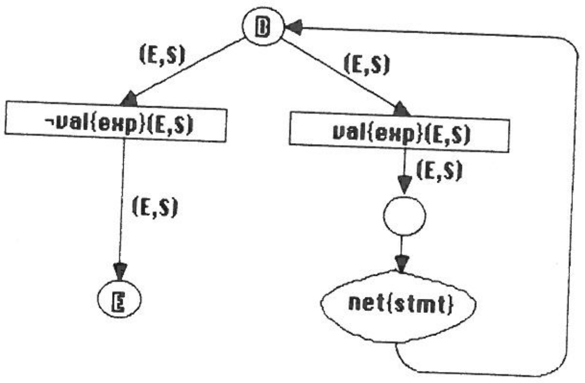
Initially the B-place of this net is marked by a token with colour ([ ],[ ]) where [ ] is the empty catalogue. All other places are initially unmarked.

4. CONTROL STRUCTURES

A <u>while statement</u> of the form

<u>while</u> exp <u>do</u> stmt

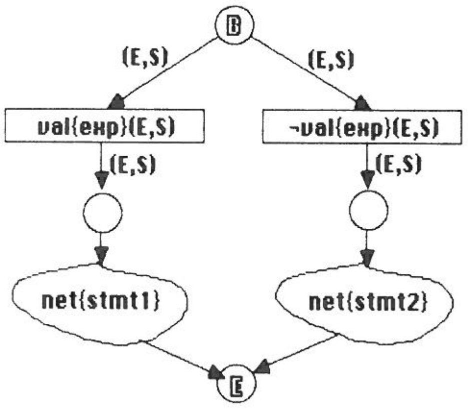
is represented by the following net



An if statement of the form

if exp then stmt1 else stmt2

is represented by the following net

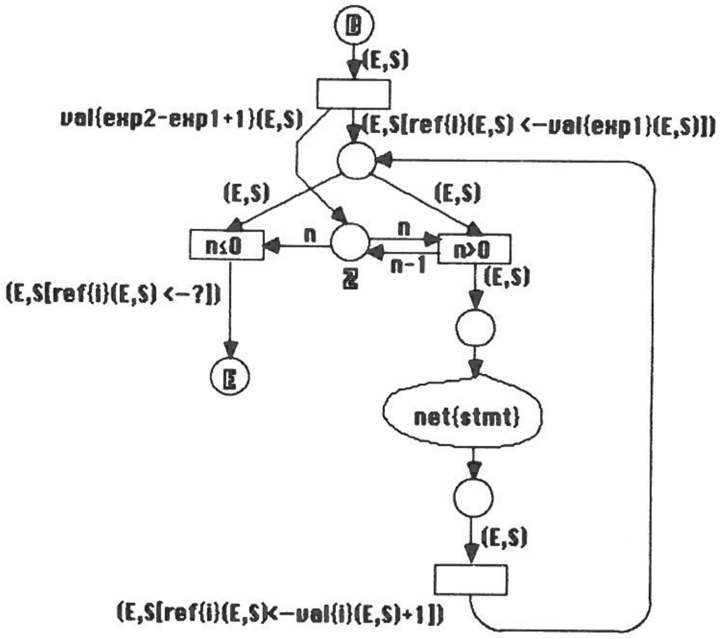


It should be obvious how to modify this to the case where the else-part is missing.

A for statement of the form

for i := exp1 to exp2 do stmt

where i, exp1 and exp2 are of the type integer, is represented by the following net.



The values of exp1 and exp2 are calculated only once, at the start of the execution of the for-loop. The number of remaining rounds in the loop is kept in the place with colour set Z. When execution finishes, i is assigned the value ?, to indicate that the value of the iteration-variable is undefined upon exit from the loop.

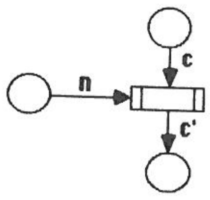
We have ignored the requirement that inside the body of the loop, i may not occur on the lefthandside of := or be passed as a var-parameter to a procedure. This can be handled by binding i directly to its value in the environment E, and making the function ref{i} undefined in such cases. (Names for constants, introduced in constant definitions,can be handled analogously.)

It should be obvious how to modify the net to the case where i, exp1 and exp2 have another type, and to the case where "to" is substituted by "downto". It is also easy to define the semantics of a repeat statement and of a case statement.

5. PROCEDURES

In order to treat recursion and procedures as parameters in a proper way, we shall extend the standard Petri net formalism as follows.

Firstly, rather than considering a single net, we shall work with a (dynamically varying) set of (uniquely) named nets. Secondly we shall introduce so-called meta-transitions which are drawn as follows.



They take as input a net-name n and an ordinary token c, and deliver as output an ordinary token c'. The firing rule of the transition is defined by means of the firing sequence(s) of n. To calculate c' we consider the net n, with the B-place marked by c. We construct the possible firing sequence(s), and define c' to be the colour of a token which eventually appears at the E-place (if such a token never appears, the meta-transition cannot fire).

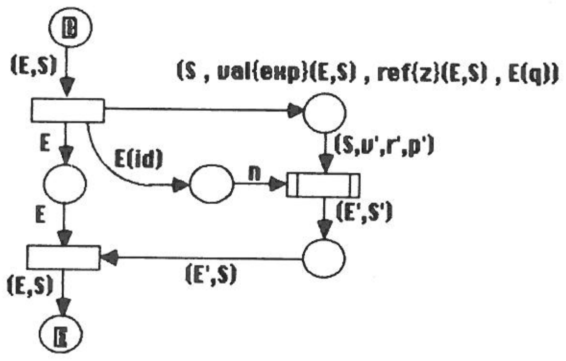
For the sake of simplicity we shall only consider procedures with exactly one value-parameter, one var-parameter, and one procedure-parameter. The syntax of call and declaration is as follows.

```
CALL:      id(exp,z,q)
DEC:       procedure id(fv:tv; var fr:tr; procedure fp(...));
               BLOCK
```

where BLOCK is a sequence of declarations followed by a sequence of statements (see section 3).

The semantics of the procedure call is represented by the following net

net{CALL}:



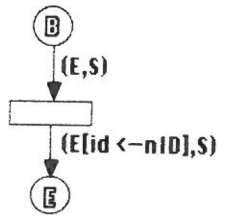
The upper transition calculates the value of the value-parameter, the address of the var-parameter and the net-name of the procedure-parameter. It also calculates the net-name of the net, which represents the procedure id itself. This net is constructed when id is declared, and it represents the actions which take place during the execution of id.

When the parameters and E(id) have been calculated in the environment of the call, this environment is put in a special place, where it is preserved. Next the procedure is executed (represented by the meta-transition) and finally the lower transition of net{CALL} re-establishes the old environment E and discards the environment E' of the procedure.

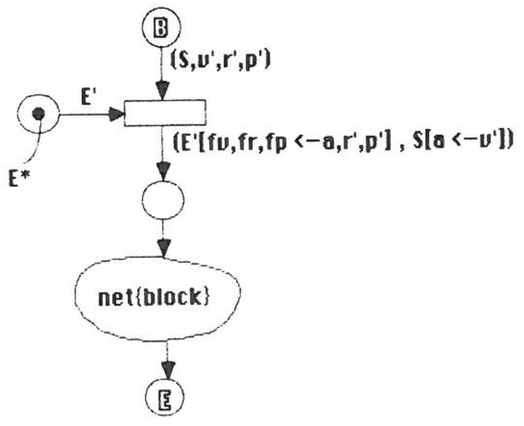
As described above, the firing rule of the meta-transition is defined by means of the firing sequence in the net n = E(id). To calculate the output (E',S') we mark the B-place by (S,v',r',p'), construct the possible firing sequence of n, and define (E',S') to be the token-colour eventually appearing at the E-place.

The semantics of the procedure declaration is represented by the following net

net{DEC}:



where nID is a (new) net-name for the following net nID:



a is a new address not appearing in S and the upper leftmost place is marked by a token representing the environment E*, which existed immediately after the procedure declaration (i.e. E* = E[id ← nID], created by the transition in net{DEC}).

The net nID remembers, by means of E*, the environment in which it was declared. In particular, this environment contains the name of the procedure itself, and this makes it possible to handle recursive calls. The upper transition of nID receives the actual parameters (which were calculated by the procedure call) and it binds them to the formal parameters in the environment E*. The value-parameter is bound to a new address, which is bound to the value of the corresponding actual parameter. The var-parameter is bound to the address of the corresponding actual parameter. The procedure-parameter is bound to the net-name of the corresponding actual parameter. Next the procedure block is executed, represented by net{BLOCK}.

Each execution of a procedure declaration introduces a new named net, representing that procedure. As mentioned above, we shall assume that all net-names are distinct.

It should be obvious how to modify the nets above to the cases where id has a different number of parameters.

We now explain more carefully why we have extended the Petri net formalism by introducing meta-transitions. There are two reasons. First of all id may be a recursive procedure. Then net{BLOCK} contains a copy of net{CALL} and (if we here insert nID from the very start) this net{CALL} contains a new copy of net{BLOCK}. In other words net{BLOCK} contains a copy of itself, which contains another copy, and so on, i.e. we get an infinite net. The other reason for introducing meta-transitions is that the name id appearing in the procedure call is not necessarily declared as a procedure. Instead it may be a formal parameter inside another procedure id'. In that case E(id) must represent the corresponding actual parameter. E(id) may change from one execution of id' to the next. Thus it cannot be calculated and inserted until the actual call is performed.

Technical remark: We have considered at least two alternative ways of handling recursion and procedures-as-parameters. The first is to define the nets recursively, i.e. to define the net corresponding to a program as a component of the minimal fixed point of a set of net-equations. As mentioned above this would imply that the nets become infinite, but the real problem is that because of the way we treat the static scope rules, the set of equations would either be infinite (recursive procedures can have local procedures) or the net would contain tokens in which names were bound to nets which are isomorphic to (parts of) the whole net. In either case the machinery becomes quite involved and it is even doubtful that the mathematics, in terms of the underlying domain theory, works out. The second alternative is to produce one net for every static procedure declaration and then handle all dynamic aspects via tokens whose colours represent stacks, dynamic and static chains etc. This would however make the approach much more implementation-oriented and would represent a definite drawback in terms of the general goal, to teach semantics.

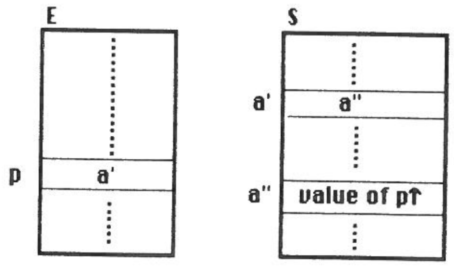
6. POINTER VARIABLES

When a pointer variable has been declared as[†]

$$p: \uparrow t$$

we can speak of the dynamic variable $p\uparrow$, which is of type t.

The value of p is an address a", and the word identified by a" contains the value of $p\uparrow$.



This calls for a generalization of the functions ref and val (defined in section 2) so they can also be used to calculate the address and value for a dynamic variable $p\uparrow$

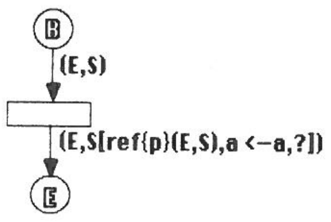
$$\begin{aligned} \text{ref}\{p\uparrow\}(E,S) &= S(\text{ref}\{p\}(E,S)) \\ \text{val}\{p\uparrow\}(E,S) &= S(\text{val}\{p\}(E,S)) \end{aligned}$$

These recursive definitions also work in the case where t is itself a pointer type.

A dynamic variable is created by means of the statement

new(p)

which we shall handle as a special kind of statement, even though it is a call of a standard procedure. The semantics is defined by the following net



where a is a new address not appearing in S. The address of p is bound to a new unused address, which in turn is bound to the ?-value.

† for the sake of simplicity we only consider the case where t is a simple type, i.e. boolean, character, integer, real or pointer.

new is a standard procedure in PASCAL, and thus it is considered to be implicitly defined prior to the program-text. We could handle standard procedures in exactly the same way as programmer defined procedures (using nets of the form net{CALL} and nID from section 5). Then we would have to modify the net of a program (see section 3), so that the B-place contained a token with colour (E_0,[ ]), where E_0 is an environment relating the name of each standard procedure to the corresponding net. This approach would however yield unnecessarily complicated nets, due to the parameter passing mechanism in net{CALL} and nID (see section 5). Thus we shall instead use the more succint net-representation given above. Similar remarks will apply to the semantics of the standard procedures read and write (see section 8).

Notice that now we cannot allow a program to contain redefinitions of standard procedures. If one wants to preserve this ability, the standard procedures should be treated in the same way as programmer defined procedures.

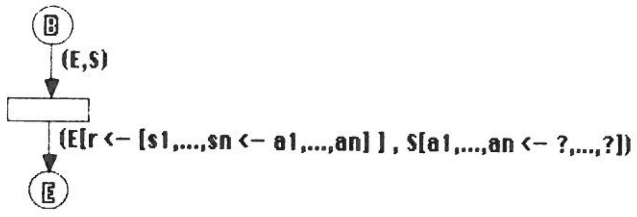
7. RECORD VARIABLES

In this section we shall discuss how to modify variable declarations (from section 3) to the case of record types.

A record declaration of the form (where t1,...,tn are assumed to be simple)

```
            r: record
                  s1: t1;
                   :                 (n ≧ 1)
                  sn: tn
               end
```

is represented by the following net


B
(E,S)
(E[r <– [s1,...,sn <– a1,...,an]], S[a1,...,an <– ?,...,?])
E

where a1,...,an are new addresses not appearing in S. The notation

[s1,...,sn ← a1,...,an]

represents a catalogue in which each selector-name si is bound to the address ai. The variable name r is bound not to an address, but to an environment, and this environment binds each selector name to the address associated with its value.

Once more we have to generalize the functions ref and val (defined in section 2 and modified in section 6) such that they can take "names" of the form r.s as arguments.

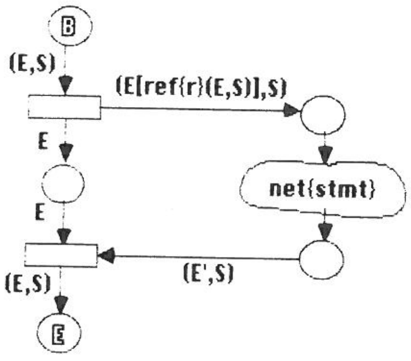
ref{r.s}(E,S) = ref{s}(ref{r}(E,S),S)
val{r.s}(E,S) = val{s}(ref{r}(E,S),S)

To calculate the address and value of a "selector variable", we do not use the normal environment E, but the special environment ref{r}(E,S) binding selector names to addresses.

The semantics of a <u>with statement</u> of the form

<u>with</u> r <u>do</u> stmt

can now be defined by the following net



where the notation E[ref{r}(E,S)] represents the catalogue obtained from E by "adding" the bindings of ref{r}(E,S) (see section 2).

8. INPUT AND OUTPUT

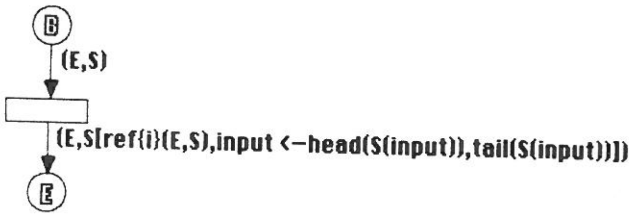
In this paper we shall deal with input and output in a very simplified (and admittedly unsatisfactory) manner. We only consider communication with the standard files input and output by means of the standard procedures read and write. Moreover we shall assume that the input file always has sufficiently many input values of correct types.

To handle input and output we shall assume the store S to have two special "addresses", input and output, each containing a sequence (of integers, booleans, reals, etc.). The B-place of net{PROG} (see section 3) now contains a token with colour ([ ] , [input,output←INPUT,Ø]), where INPUT is the contents of the input file when program execution starts, while Ø is the empty sequence.

A <u>read command</u> of the form

read(i)

is represented by the following net.

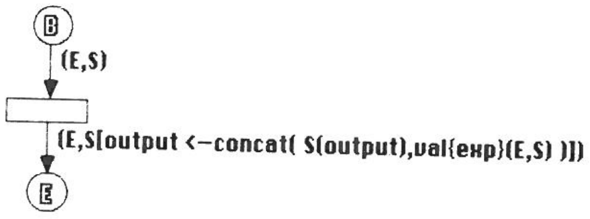


The first element of the input file is removed and its value bound to the address of i.

A <u>write command</u> of the form

write(exp)

is represented by the following net.



The value of exp is concatenated to the contents of the output file.

It should be obvious how to extend the semantics above to the case where more variables/expressions are input/output by the same read/write command. It is also straightforward to describe the semantics of readln and writeln.

9. EXAMPLES

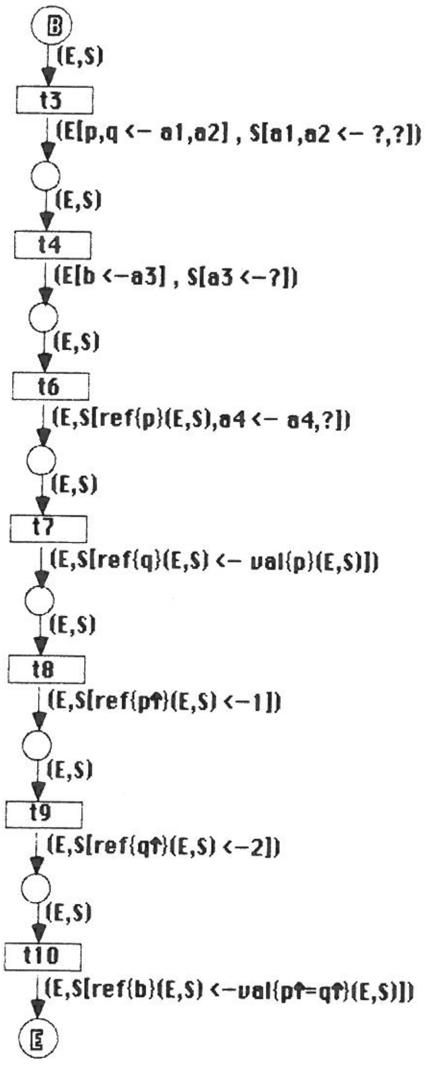
This section contains three examples showing how nets are constructed for small Pascal programs.

Example 1

This example illustrates the semantics of variable declarations and assignment statements.

```
 1 PROGRAM CONFLICT(INPUT,OUTPUT);
 2   VAR
 3     P,Q: ↑INTEGER;
 4     B:    BOOLEAN;
 5   BEGIN
 6     NEW(P);
 7     Q:=P;
 8     P↑:=1;
 9     Q↑:=2;
10     B:=(P↑=Q↑)
11   END (*CONFLICT*).
```

The program is represented by the following net, where transition ti represents program line no. i.


B
(E,S)
t3
(E[p,q <- a1,a2], S[a1,a2 <- ?,?])
(E,S)
t4
(E[b <-a3], S[a3 <-?])
(E,S)
t6
(E,S[ref{p}(E,S),a4 <- a4,?])
(E,S)
t7
(E,S[ref{q}(E,S) <- val{p}(E,S)])
(E,S)
t8
(E,S[ref{p↑}(E,S) <-1])
(E,S)
t9
(E,S[ref{q↑}(E,S) <-2])
(E,S)
t10
(E,S[ref{b}(E,S) <-val{p↑=q↑}(E,S)])
E

The firing sequence of the net can be represented by the following state-schemata, where each column describes the changes in E and S made by a transition:

			t3	t4	t6	t7	t8	t9	t10
E	p		a1						
	q		a2						
	b			a3					
S	a1		?		a4				
	a2		?			a4			
	a3			?					true
	a4				?		1	2	
	input	∅							
	output	∅							

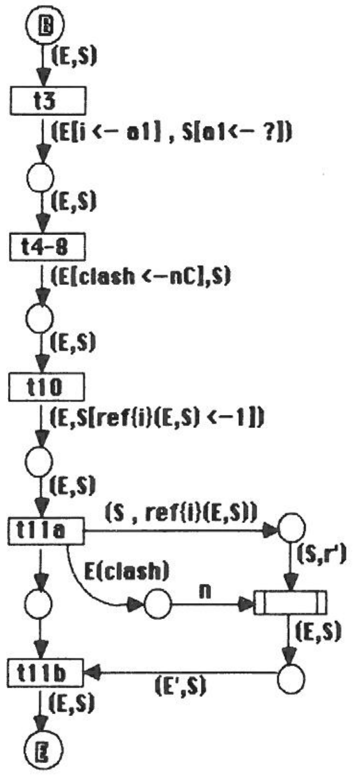
Variable names/addresses enter the catalogue when they are given their first address/value. As an example, b enters E and a3 enters S when transition t4 fires.

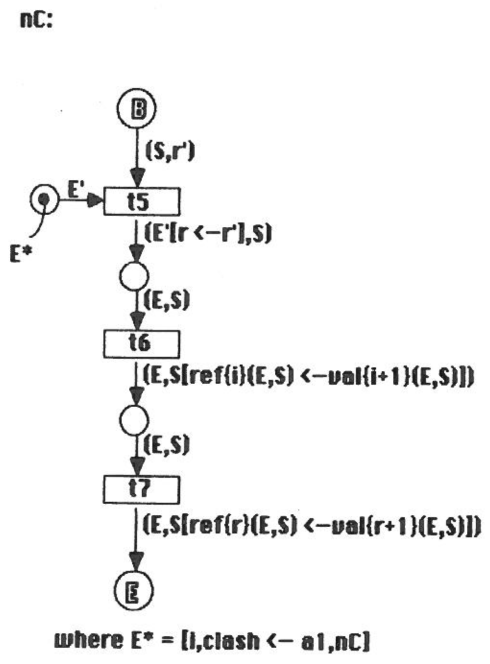
Example 2

This example illustrates the semantics of var-parameters.

```
PROGRAM SHARING(INPUT,OUTPUT);
  VAR
    I: INTEGER;
  PROCEDURE CLASH(VAR R: INTEGER);
    BEGIN
      I:=I+1;
      R:=R+1
    END (*CLASH*);
  BEGIN
    I:=1;
    CLASH(I)
  END (*SHARING*).
```

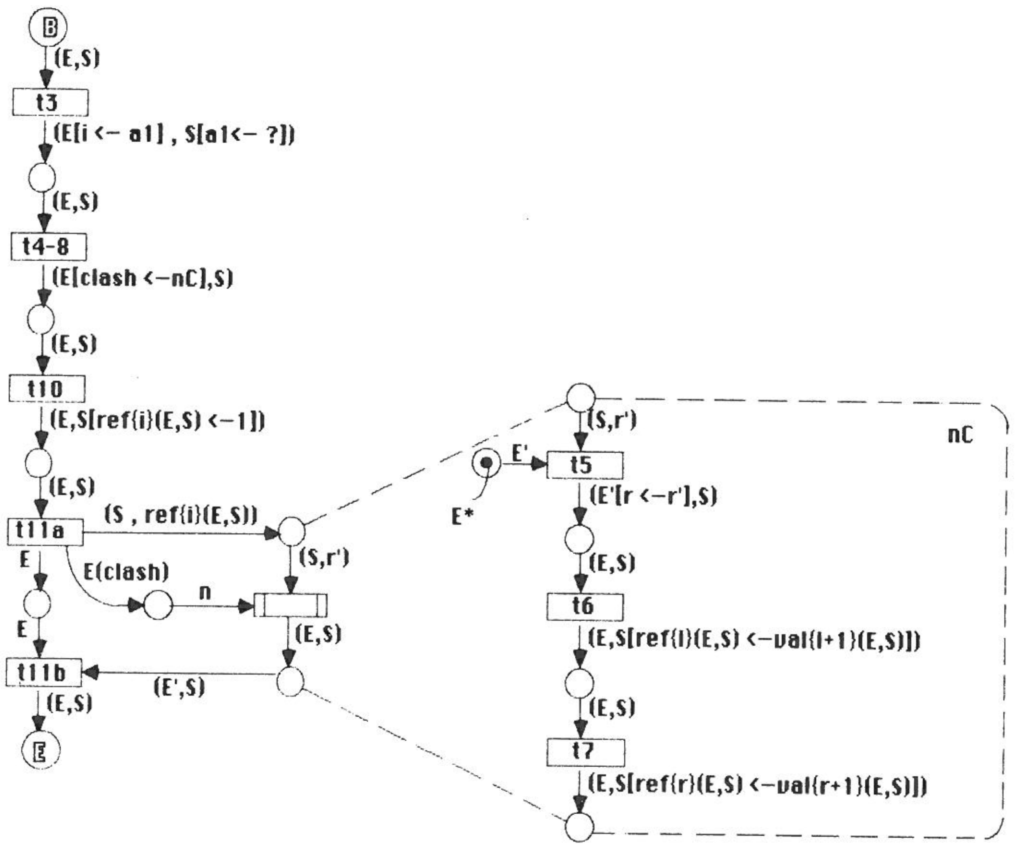
The program is represented by the following two nets:




nC:
B
(S,r')
E'
t5
E*
(E'[r <-r'],S)
(E,S)
t6
(E,S[ref(i)(E,S) <-val(i+1)(E,S)])
(E,S)
t7
(E,S[ref(r)(E,S) <-val(r+1)(E,S)])
E
where E* = [i,clash <- a1,nC]

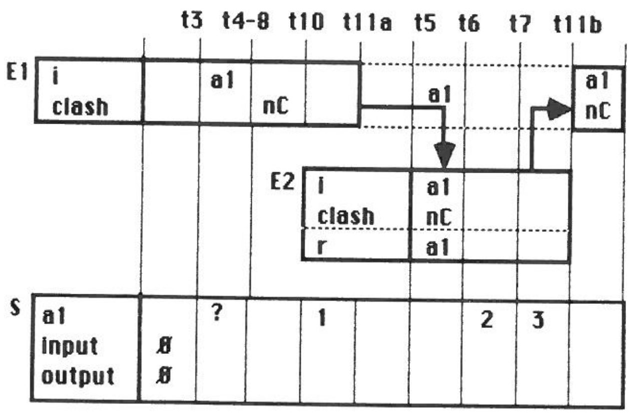
The leftmost net represents the main program, while the rightmost net (with the net-name nC) represents the procedure clash.

When transition t4-8 fires, the net-name nC is bound to the name clash and when transition t11a fires, the net-name nC is put on one of the input places of the meta-transition. This means that the effect of the meta-transition is determined by a firing sequence in the net named by nC. We shall indicate this by the following graphical notation:



where E* = [i,clash <- a1,nC]

The firing sequence of the net can be represented by the following state-schemata.



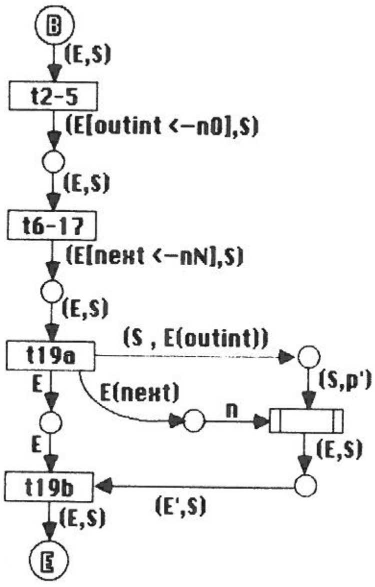
When procedure clash is evoked by transition t11a a new environment E2 is created. It consists of the environment E* which existed when clash was declared, augmented by the bindings for the formal parameters of clash. In the state-schemata, the two parts of E2 are separated by a horizontal dotted line. The environment E1 of the call is preserved by transition t11a and re-established by t11b. The position of the two bending arcs of the state-schemata indicate that clash is being called, and the inscription on the leftmost arc indicates that a1 is passed as actual parameter.

Example 3

This example illustrates the semantics of standard procedures, recursive calls, and procedure-parameters. It shows how the combination of static scope rules and procedures-as-parameters can be used to traverse the runtime stack. The example is considerably more complicated than the two earlier ones:

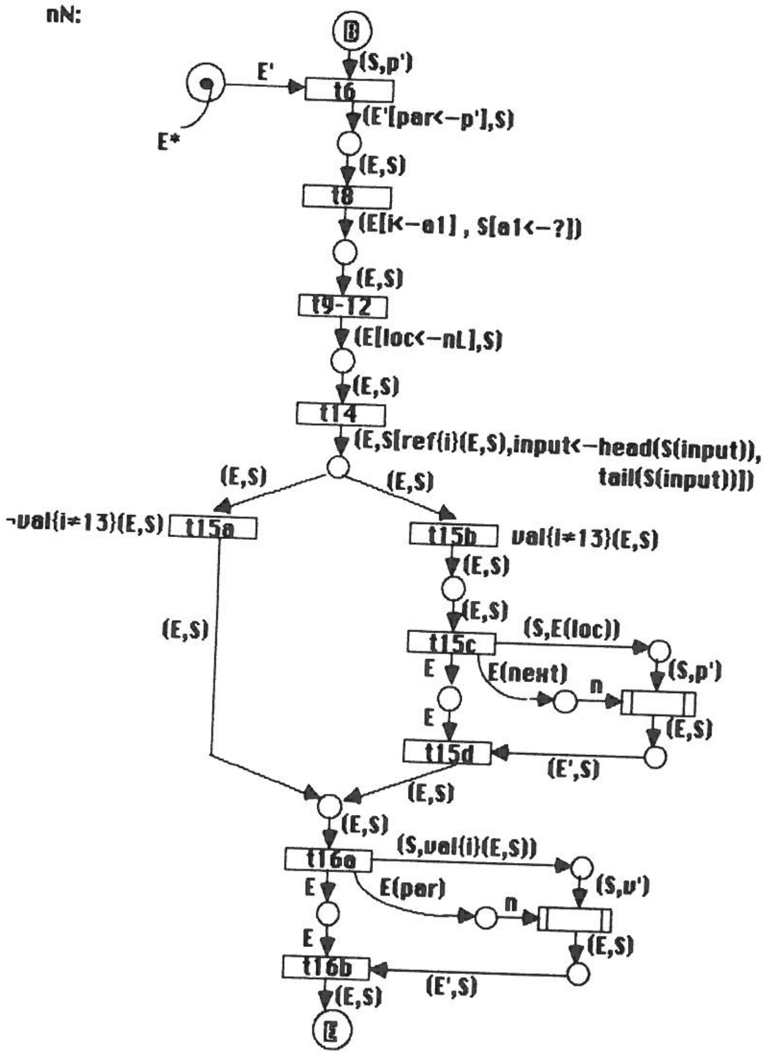
```
 1 PROGRAM ADD(INPUT,OUTPUT);

 2   PROCEDURE OUTINT(N: INTEGER);
 3     BEGIN
 4       WRITE(N)
 5     END (*OUTINT*);

 6   PROCEDURE NEXT(PROCEDURE PAR(F: INTEGER));
 7     VAR
 8       I: INTEGER;
 9     PROCEDURE LOC(F: INTEGER);
10       BEGIN
11         I:=I+F
12       END (*LOC*);
13     BEGIN
14       READ(I);
15       IF I<>13 THEN NEXT(LOC);
16       PAR(I)
17     END (*NEXT*);

18   BEGIN
19     NEXT(OUTINT)
20   END (*ADD*).
```

The program is represented by the following net.



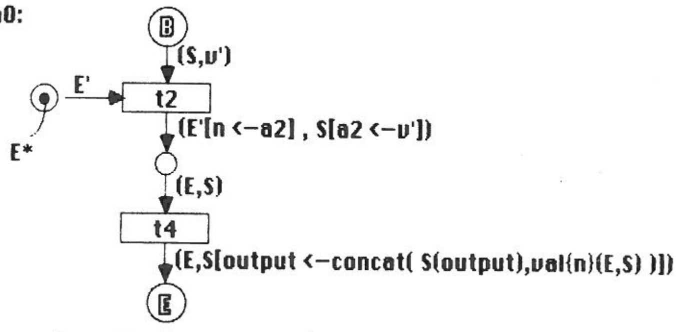
where nN, nO and nL are net-names for the following nets (representing next, outint and loc).

nN:



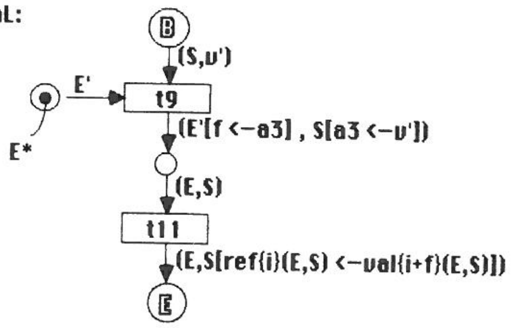
where E* = [outint,next <-n0,nN]

n0:


B
(S,ʋ')
E'
t2
E*
(E'[n <–a2] , S[a2 <–ʋ'])
(E,S)
t4
(E,S[output <–concat(S(output),ʋal(n)(E,S))])
E

where E* = [outint <–n0]

nL:

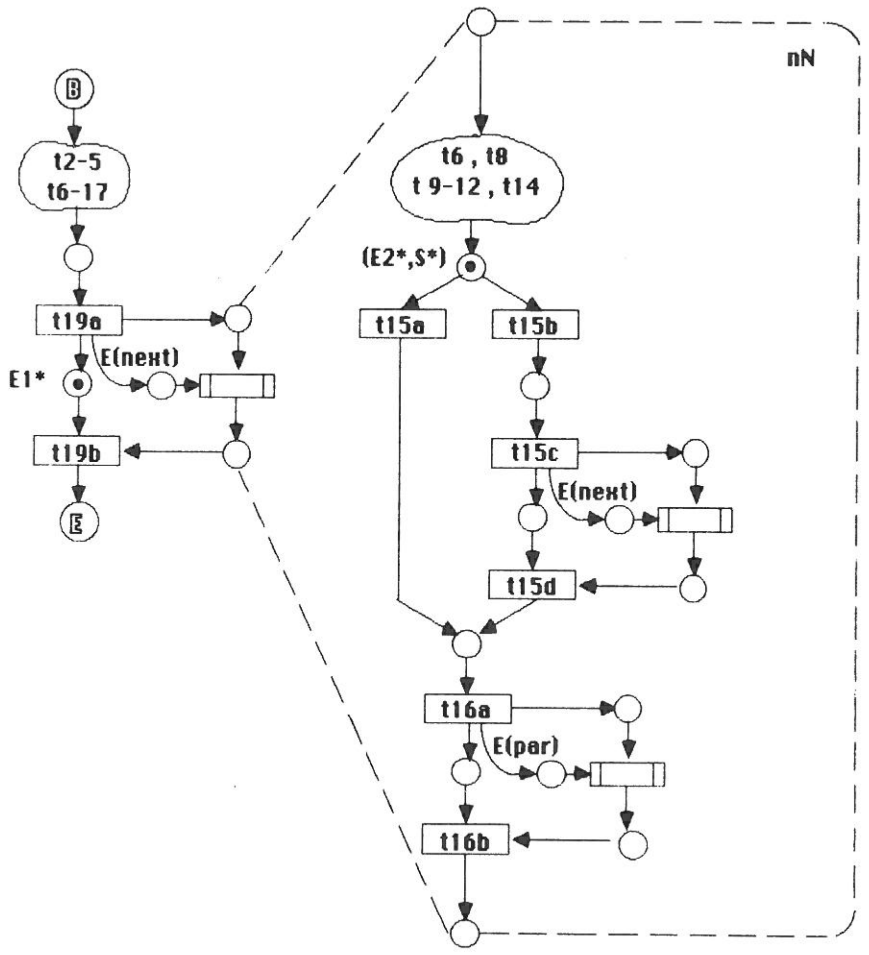


where E* = [outint,next <–n0,nN] [par,i,loc <–n0,a1,nL]

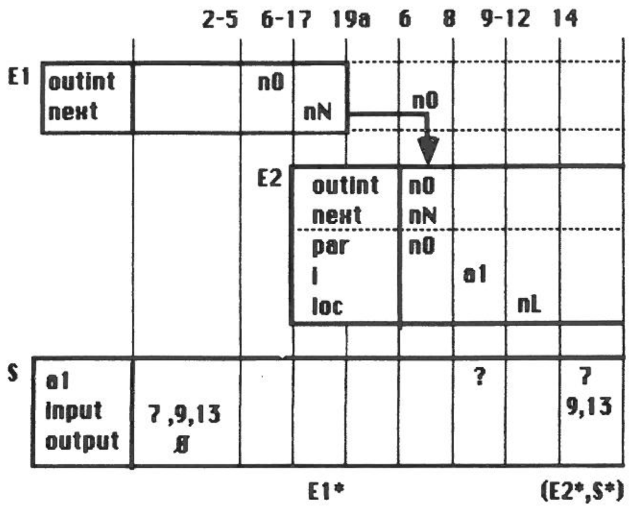
We now consider an execution of the program add, in which the input file contains the following numbers

7, 9, 13

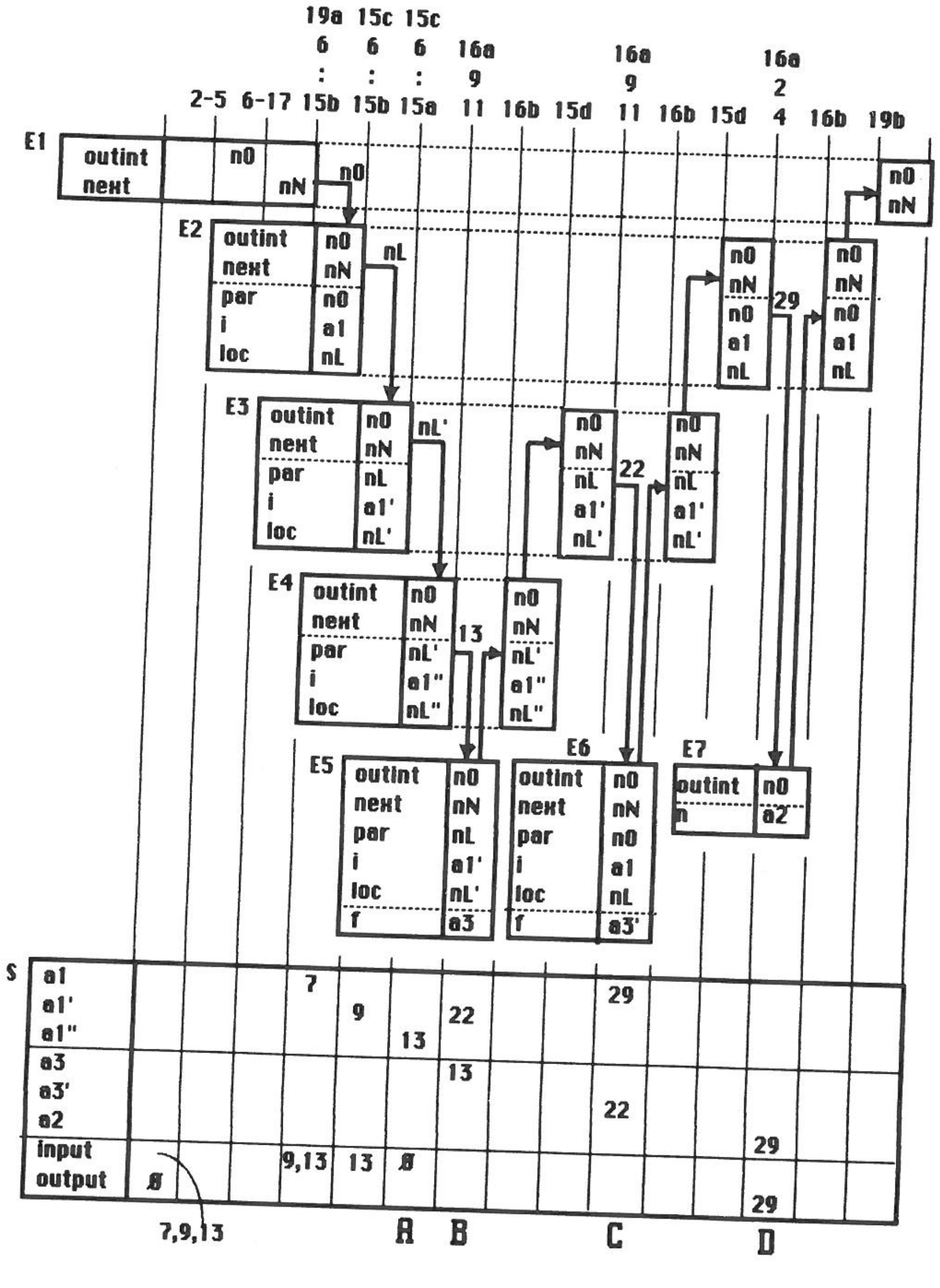
Immediately after the reading of 7, represented by transition t14, we have the following net



where the token-colours E1* and (E2*,S*) are defined by the following state-schemata.

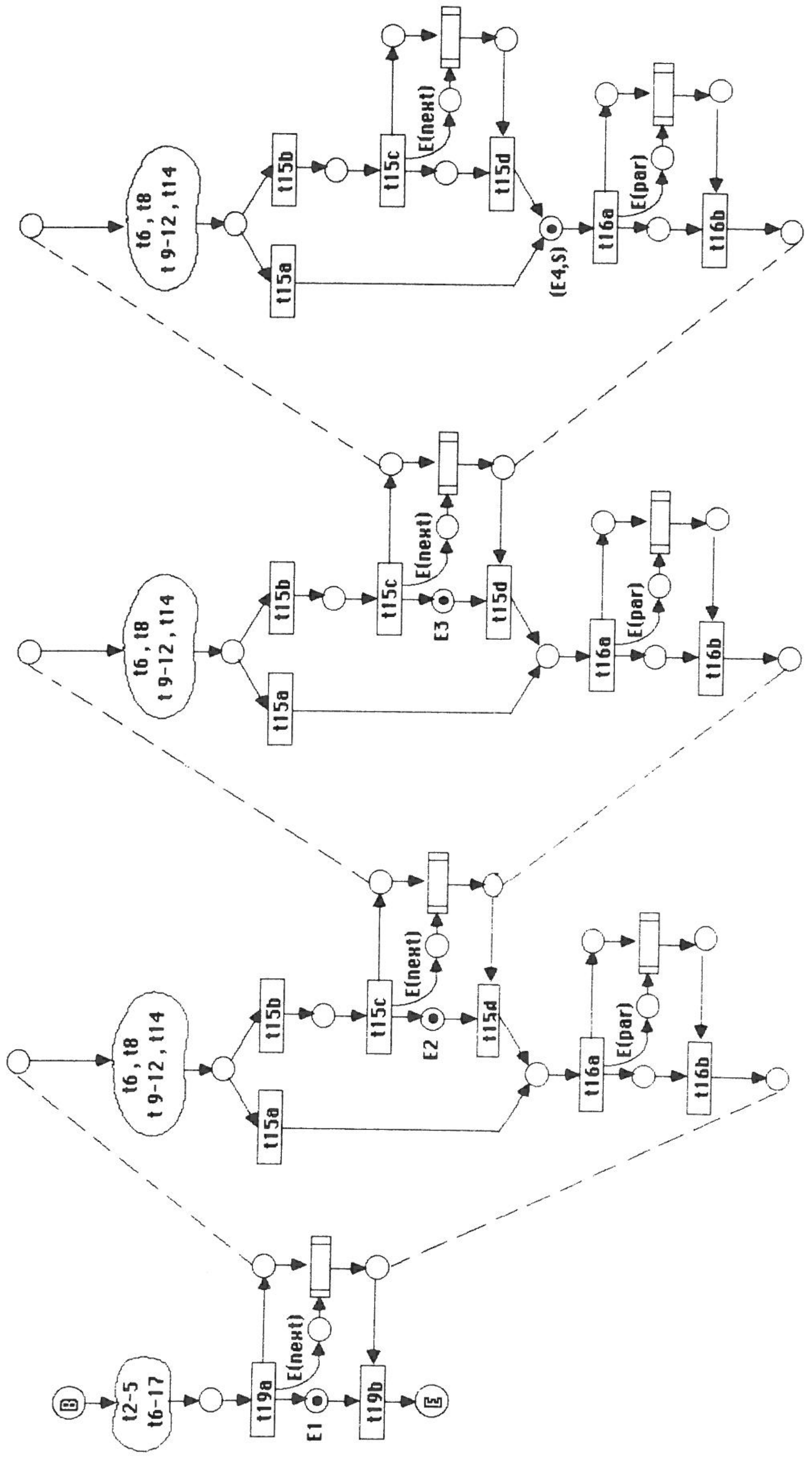


The first incarnation of next has read 7 from the input file, and since 7 < > 13 transition t15b is chosen. This means that we get a recursive call of next. The second incarnation of next reads 9, and since 9 < > 13 we get a third incarnation, which reads 13. Now transition t15a is chosen and we have the net shown on the next page. The token-colours are determined by column A in the following (condensed) state-schemata. The nets nL, nL', and nL" represent the three different incarnations of the loc procedure, defined in the three different incarnations of next. These three nets are identical except for the E* catalogue, which is E2, E3 and E4, respectively.



19a 15c 15c 16a 16a 16a
2-5 6-17 15b 15b 15a 11 16b 15d 11 16b 15d 4 16b 19b
E1 outint next n0 nN
E2 outint next par i loc n0 nN n0 a1 nL
E3 outint next par i loc n0 nN nL a1' nL'
E4 outint next par i loc n0 nN nL' a1" nL"
E5 outint next par i loc f n0 nN nL a1' nL' a3
E6 outint next par i loc f n0 nN n0 a1 nL a3'
E7 outint n n0 a2
13 22 29
S a1 a1' a1" a3 a3' a2 input output
7 9 13 22 29 13 22 29
9,13 13 Ø 29 Ø 29
7,9,13
A B C D

We are now ready to perform the procedure-call par(i) in the third incarnation of next. The net of the procedure is evaluated in E4, where


B
t2-5
t6-17
t19a
E(next)
E1
t19b
E
t6, t8
t 9-12, t14
t15a
t15b
t15c
E(next)
E2
t15d
t16a
E(par)
t16b
t6, t8
t 9-12, t14
t15a
t15b
t15c
E(next)
E3
t15d
t16a
E(par)
t16b
t6, t8
t 9-12, t14
t15a
t15b
t15c
E(next)
t15d
(E4,S)
t16a
E(par)
t16b

par is bound to nL' (the loc procedure declared in the second incarnation of next; this is declared in E3 and thus E5 (upper part) becomes equal to E3). The actual parameter i is also evaluated in E4. It has the value 13, which is bound to the formal parameter f, via the address a3. The statement i := i+f is executed in E5, where i is bound to a1' (with value 9) while f is bound to a3 (with value 13). Thus i gets the value 22 (via the address a1'), and we have the situation depicted by column B.

We now return from the call par(i) and then from the third incarnation of next to the second, which is ready to perform the procedure call par(i). The net of the procedure is evaluated in E3, where par is bound to nL (the loc procedure in the first incarnation of next; this is declared in E2, and thus E6 (upper part) becomes equal to E2). This procedure is called with 22 as parameter-value. The statement i := i+f now involves the addresses a1 and a3', and the former gets the value 29. This is depicted by column C.

Once more we return from the call par(i) and then from the second incarnation of next to the first, which is ready to perform the procedure call par(i). The net of the procedure is evaluated in E2, where par is bound to nO (the outint procedure; this is declared in E1, and thus E7 (upper part) becomes equal to E1 (as it was when outint was declared)). This procedure is called with 29 as parameter-value and thus the output file receives this value, via the write statement (transition t4). This is depicted by column D.

Again we return from the call par(i) and then from the first incarnation of next to the main program, which finishes the execution.

10. EXERCISES

This section contains typical examples of exercises.

1. Construct a net, which defines the semantics of a repeat statement of the form

   repeat stmt until exp

2. Construct a net, which defines the semantics of a case statement of the form

```
case exp of
   const_1 : stmt_1;
     :         :                 (n ≧ 1)
   const_n : stmt_n
end
```

3. Construct the two nets defining the semantics of the following program, which calculates the n'th Fibonacci-number by means of a recursive procedure.

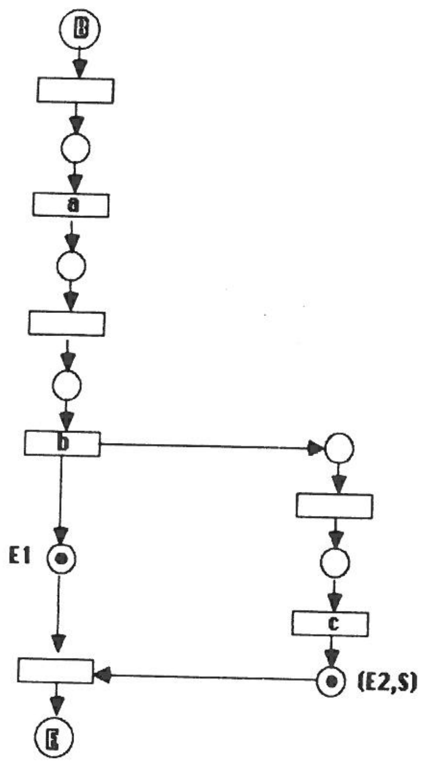
```
program fibonacci(input ,output);
  var
    i,r:integer;
  procedure fib(n:integer; var r:integer);
    var
      k:integer;
    begin
      if n<=2 then
        r:=1
      else
      begin
        fib(n-1,r);
        fib(n-2,k);
        r:=r+k
      end
    end (*fib*);
  begin
    readln;
    read(i);
    fib(i,r);
    writeln(r)
  end (*fibonacci*).
```

What does the program-net look like just before statement "r:=r+k" in the body of fib is executed for the first time? What is the marking?

4. The Pascal program

```
program test(input ,output);

  var
    x:integer;
    r:record
        x:integer;
        y:integer
      end;
  begin
    x:=0;
    with r do
    begin
      x:=1;
      r.y:=2
    end
  end (*test*).
```

is represented by the following net where the marking represents the state immediately after the execution of the statement "r.y:=2".



What are the arc-inscriptions which surround transitions a, b, and c? What are the contents of the catalogues E1, E2 and S?

5. Assume that Pascal is modified in such a way that procedures in addition to value-parameters, var-parameters, and procedure-parameters, also may have result-parameters (i.e. parameters where the actual parameter is assigned the value of the formal parameter when the procedure returns). Discuss how net{CALL} and nID in section 5 must be modified if the procedure ID also has a result parameter. It is only necessary to change the arc-inscriptions, while the net-structure is unaltered.

6. Discuss how the semantics of arrays, sets and scalar types can be defined. There are several possible ways to do this, and the purpose of this exercise is to sketch and compare some of these alternatives.

11. CONCLUSION

The material presented in this paper has been used, with only small modifications, as part of the teaching material for the introductory computer science course at Aarhus University since 1980.

The starting point of its development was a decision that time had come where not only syntax, but also semantics, should be taught in a systematic and precise way to first year students.

Given this decision, the question was what semantic tool to use. The main reason for choosing a Petri net based formalism were the following.

- Petri nets are also (even better) suited for describing languages with concurrency. Thus the approach generalizes to such languages, and the students don't have to learn too many formalisms.
- It was believed that the graphical representation of the nets, together with their operational flavour, would support intuition and thereby make the semantics relatively easy to understand.

These expectations have essentially been met. The details of the teaching approach are as follows.

First the students are given an informal explanation of the different language constructs in PASCAL, and they get some experience using the language to create small programs. Then the semantics is introduced and used to explain the more complicated language features such as var-parameters, procedure-parameters, scope rules, dynamic variables, etc.

Earlier in the same course the students are given an informal introduction to high-level Petri nets, which is used to describe the semantics of a small system description language. Thus the students already know the basic ideas of this type of semantic description and it is rather easy for them to learn the PASCAL semantics.

Altogether 8 lectures of 2×45 min. are used: one lecture to introduce high-level Petri nets, two lectures for our semantic model and the semantics of the small system description language, three lectures for the informal explanation of PASCAL, and then only two lectures for the PASCAL semantics itself. It is our opinion that this approach has not required more lecturing time than needed for an equally detailed informal explanation of the PASCAL semantics.

Our experience tells us that indeed it is quite easy for the students to learn the semantics in this way. In particular the semantics of declarations, control structures, parameter mechanisms, recursion and scope rules come across very nicely. If the students have difficulties, it is because of the rather compact notation for inscriptions which has been borrowed from denotational semantics. In fact it turns out that some students understand the main ideas of the semantics from its graphical representation, even though they fail to understand the formal firing rules of Petri nets and the formal rules for manipulating environment and store in denotational semantics.

It might be argued that we have only treated a modest subset of PASCAL, without considering type definitions and functions, and without covering the file-, array- and set-types. It is however (with one exception) quite straightforward to extend the semantics to include these aspects.

The meaning of a <u>type definition</u> can be defined by binding an allocation function to the type name (in the environment E). The allocation function is invoked when the type name appears on the righthandside of ':' in variable declarations and parameter lists.

Function <u>declarations</u> are treated in essentially the same way as procedure declarations. Function <u>calls</u> are syntactically (part of) expressions, and in order to handle this, it is convenient to introduce explicit expression evaluating nets everywhere in the semantics. For an expression, exp, without function calls, this net is just a simple transition which evaluates val{exp}(E,S). For an expression with function calls, the net is built from nets representing the functions. The way these nets are composed is determined by the structure of the expression in the usual way.

The handling of arrays, sets and scalar types is more or less an exercise in denotational semantics, and files can be treated in much the same way as input/output in section 8.

There remains however one problem: to treat communication, including external files, in a satisfactory way. The inclusion of external files as special entries in the store comes very close to cheating, and it certainly does not explain the notion of <u>interactive</u> programs. This type of counter-intuitive "trick" is not to be recommended, particularly not for beginners, and a nice way of handling this problem would be very welcome.

Acknowledgements

Horst Oberquelle provided lots of helpful suggestions. The remarks of 4 anonymous referees are also acknowledged.

References

[1] H.J. Genrich and K. Lautenbach: System modelling with high-level Petri nets, Theoretical Computer Science 13 (1981), 109-136.

[2] U. Goltz and W. Reisig: CSP-programs as nets with individual tokens. G. Rozenberg (ed.): Advances in Petri Nets 1984, Lecture Notes in Computer Science 188, Springer-Verlag 1985, 169-196.

[3] Ka. Jensen and N. Wirth: Pascal user manual and report. Springer-Verlag 1975.

[4] K. Jensen: Coloured Petri nets and the invariant-method. Theoretical Computer Science 14 (1981) 317-336.

[5] K. Jensen: High-level Petri nets. A. Pagnoni and G. Rozenberg (eds.): Applications and Theory of Petri Nets, Informatik-Fachberichte 66, Springer-Verlag 1983, 166-180.

[6] K. Jensen: An introduction to high-level Petri nets. Proceedings of the ISCAS 85 Conference, Kyoto, Japan 1985, IEEE, 723-726.

[7] K. Jensen, M. Kyng: Epsilon. A system description language. Computer Science Department, Aarhus University, DAIMI PB-150, 1982.

[8] N.D. Hansen and K.H. Madsen: Formal semantics by a combination of denotational semantics and high-level Petri nets. A. Pagnoni and G. Rozenberg (eds.): Applications and Theory of Petri Nets, Informatik-Fachberichte 66, Springer-Verlag 1983, 132-148.