Knowledge Representation with Attribute Grammars

G. PAPAKONSTANTINOU

National Technical University of Athens, Electrical Engineering Department, Computer Science Division, Athens 15773, Greece

J. KONTOS

Computers Department, N.R.C. 'Democritos', Aghia Paraskevi, Athens, Greece

The use of attribute grammars for knowledge representation is examined in the present paper. It is shown how data knowledge and knowledge-base knowledge can be represented using syntactic and semantic notation. Control knowledge is represented by the parsing mechanism of the interpreter used. It is proposed that attribute grammar evaluators may prove to be useful knowledge engineering tools.

Received July 1984

1. INTRODUCTION

Attribute grammars were devised by Knuth as a tool for formal language specification.[1] Noonan[2] has proposed that attribute grammars can be used for the formal specification of string processing problems. The use of attribute grammars as a programming language is studied in Hehner and Silverberg.[3] Attribute grammars have been extended in Watt and Madsen[4] in order to facilitate the formal specification of programming languages. The successful application of an attribute grammar interpreter that was reported in Papakonstantinou,[6] to the programming of the solution of problems as diverse as waveform analysis,[6] plan recognition,[5,7] interpreter description,[7] filtering[9] and sentence or pattern generation[10,11] has prompted us to investigate the feasibility of using attribute grammars for knowledge representation. The results of our study are reported in the present paper, in which we show how knowledge representation can be accomplished using attribute grammars and how inferences can be drawn from such knowledge using an attribute grammar interpreter. It is thus proposed that attribute grammars for which efficient evaluators do exist may be a useful knowledge engineering tool.

Knowledge representation is the main activity distinguishing 'expert systems' from other application computer systems.[12] Ordinary computer systems organize knowledge on two levels, i.e. data and program. Most expert computer systems, however, organise knowledge on three levels, i.e. data, knowledge base and control.

At the data level declarative factual knowledge is represented. For the representation at this level the tools used include first-order predicate logic formulas, semantic networks and frames. At the knowledge-base level inferential knowledge is represented which is necessary for the deduction of new facts not included in the data level. The main tools used at this level are logic programming (e.g. Prolog) and production systems (e.g. OPS). These two classes of tools represent the two main trends, namely the declarative and procedural approach to knowledge representation. Control knowledge is normally not available to the knowledge programmer but is offered ready made by the toolmaker. This last kind of knowledge defines the inference mechanism which is responsible for the interpretation of the other bodies of knowledge.

In this paper we show how an attribute grammar can be used to represent knowledge at both the data and the knowledge-base levels. The control knowledge is embedded in the attribute grammar interpreter.

2. ATTRIBUTE GRAMMAR NOTATION USED

In the following we shall use the notation of attribute grammars augmented with a global attribute *FLAG* which takes the values **true** and **false**.[6] When *FLAG* takes the value **false** during the semantics evaluation of a BNF rule, the parser considers that matching is not successful. Hence parsing can be directed by the semantics. The class of attribute grammars to be considered are those which can be evaluated in a single pass from left to right. Bochman[13] has given a condition for an attribute grammar which assures that the semantics can be evaluated in a single scan from left to right.

The general idea of using an attribute grammar as a knowledge engineering tool is to use only one terminal symbol, the **nil** symbol. Thus the grammar is such that it recognises only empty strings of characters. During the recognition of an input string (actually the empty string) the semantics can be such that at the time they are evaluated they perform the inferences required. Moreover, relations correspond to non-terminals and their arguments to associated corresponding attributes.

3. DATA KNOWLEDGE REPRESENTATION

Data knowledge can be represented in an attribute grammar by a rule that has a left-hand part only such as:

$$\langle likes\rangle ::= \textbf{nil}$$
$$\textbf{if } x(\langle likes\rangle) \neq John \textbf{ then } FLAG := \textbf{false}; \qquad (1)$$
$$\textbf{if } y(\langle likes\rangle) \neq Mary \textbf{ then } FLAG := \textbf{false};$$

for the fact '*John likes Mary*'. The first line of this rule is the syntactic part of the rule and the other lines of the rule constitute the part that controls semantically the syntax analysis. The attributes x and y are inherited in this case, and their values which are defined by other rules are tested for conformity with the fact '*John likes Mary*'.

The question '*John likes Jenny*?' can be represented by a rule of the form:

$$\langle question\rangle ::= \langle likes\rangle$$
$$x(\langle likes\rangle) := John;$$
$$y(\langle likes\rangle) := Jenny;$$

which will give an answer **No** according to the semantics given.

If the fact that we want to represent has the form *John likes Mary's mother* then we can write:

$\langle likes \rangle ::= \textbf{nil}$

If $x(\langle likes \rangle) \neq John$ **then** $FLAG := \textbf{false}$; (2)

If $y(\langle likes \rangle) \neq mother(Mary)$ **then** $FLAG := \textbf{false}$;

where *mother* (z) is a function that given z returns the name of *z's* mother. We note that we could avoid the use of the function *mother* by using another syntactic rule for *mother* as it can be seen later on.

For a complete representation of a fact we must also examine the case of using it to answer questions other than truth or falsity such as *who likes Mary*?, *who likes John*? and *who likes who*? To face such demands we must expand the semantics of (1) as follows:

$\langle likes \rangle ::= \textbf{nil}$

If $x(\langle likes \rangle) = \textbf{nil}$ **then** $x(\langle likes \rangle) := John$;

If $y(\langle likes \rangle = \textbf{nil}$ **then** $y(\langle likes \rangle) := Mary$; (3)

If $x(\langle likes \rangle) \neq John$ **then false**;

If $y(\langle likes \rangle) \neq Mary$ **then false**;

The question *who likes Mary* may be represented as:

$\langle question \rangle ::= \langle likes \rangle$

$y(\langle likes \rangle) := Mary$;

output $(x\langle likes \rangle)$;

Output is a function for printing values of attributes.

The answer will be *John*, provided that all attributes are initialised to **nil**.

In this example the attributes x,y are used both as inherited and synthesised. It is more practical to use for each argument $t_1, t_2, \ldots, t_k$ of a relation $R(t_1, t_2, \ldots, t_k)$ two attributes, one synthesised $[a_j^S(\langle R \rangle) \quad 1 \leqslant j \leqslant k]$ and one inherited $[a_j^I(\langle R \rangle)]$.

A fact $R(c_1, c_2, \ldots, c_k)$ *is true*, where c_j, $1 \leqslant j \leqslant k$ are constants can now be represented by the rule

$\langle R \rangle ::= \textbf{nil}$

for *all* j, $1 \leqslant j \leqslant k$ **do**

if $a_j^I(\langle R \rangle) \neq c_j$ **and** $a_j^I(\langle R \rangle) \neq \textbf{nil}$ **then** $FLAG := \textbf{false}$

else, $a_j^S(\langle R \rangle) := c_j$; (4)

The fact *John likes Mary* or *likes* (*John*, *Mary*) can now be represented by the rule

$\langle likes \rangle ::= \textbf{nil}$

if $a_1^I(\langle likes \rangle) \neq John$ **and** $a_1^I(\langle likes \rangle) \neq \textbf{nil}$

then $FLAG := \textbf{false}$

else $a_1^S(\langle likes \rangle) := John$; (5)

if $a_2^I(\langle likes \rangle) \neq Mary$ **and** $a_2^I(\langle likes \rangle) \neq \textbf{nil}$

then $FLAG := \textbf{false}$ **else**

$a_2^S(\langle likes \rangle) := Mary$;

The question *who likes Mary* or *likes* (?, *Mary*) may be represented as

$\langle question \rangle := \langle likes \rangle$

$a_2^I(\langle likes \rangle) := Mary$

output $(a_1^I(\langle likes \rangle))$;

and will give the answer *John* due to the existence of rule (5).

As is illustrated on the above examples, data knowledge can easily be represented by simple attribute grammar rules that in fact contain only semantic information.

4. KNOWLEDGE-BASE REPRESENTATION

A knowledge-base consists of rules that may produce new facts that are not contained initially in the data knowledge. Such a rule may be *John greets X if he likes him* or *greets* (*John,X*) *if likes* (*John,X*) and it may be represented as:

$\langle greets \rangle ::= \langle likes \rangle$

if $a_1^I(\langle greets \rangle) \neq John$ **and** $a_1^I(\langle greets \rangle) \neq \textbf{nil}$ **then** $FLAG := \textbf{false}$;

$a_1^I(\langle likes \rangle) := John$;

$a_2^I(\langle likes \rangle) := a_2^I(\langle greets \rangle)$; (6)

$a_1^S(\langle greets \rangle) := a_1^I(\langle greets \rangle)$;

$a_2^S(\langle greets \rangle) := a_2^S(\langle likes \rangle)$;

The question *John greets Mary* or *greets* (*John*, *Mary*), may be represented as:

$\langle question \rangle ::= \langle greets \rangle$

$a_1^I(\langle greets \rangle) := John$;

$a_2^I(\langle greets \rangle) := Mary$;

output $(a_1^S(\langle greets \rangle)$, *greets*, $a_2^S(\langle greets \rangle)$;

will print *John greets Mary* due to (5) and (6).

This may be better understood with the use of a parse tree as shown in Figure 1, where the information flow is also exhibited.

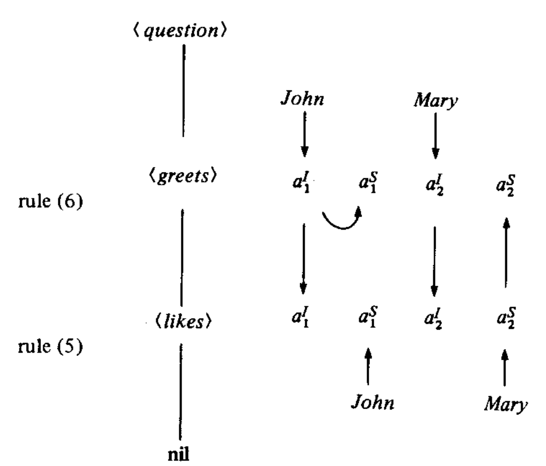

Figure 1. Parse tree of the example *John greets Mary*.

The question to *Whom greets John*? or *greets* (*John*,?) may be represented as:

$\langle question \rangle := \langle greets \rangle$

$a_1^I(\langle greets \rangle) := John$;

output $(a_2^S(\langle greets \rangle))$

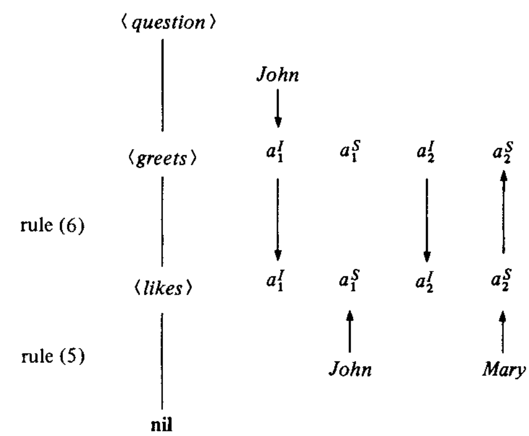

Figure 2. Parse tree of the example *Who greets John*?

and will give the answer *Mary* as it is illustrated in Figure 2.

In the general case the inference rules can be written in the form of productions in a Prolog-like notation as:

$R_0(t_{01}, t_{02}, \ldots, t_{0k_0})$ is true if

$R_1(t_{11}, t_{12}, \ldots, t_{1k_1})$ is true and

⋮

is true and

$R_m(t_{m1}, t_{m2}, \ldots, t_{mk_m})$ is true, where t_{ij}, $0 \leqslant i \leqslant m$, $1 \leqslant j \leqslant k_i$ are constants or variables.

This rule may be represented in an attribute grammar notation with the syntax rule:

$$\langle R_0 \rangle ::= \langle R_1 \rangle \langle R_2 \rangle \ldots \langle R_m \rangle. \quad (7)$$

We use k_j inherited attributes a_l^I, $1 \leqslant l \leqslant k_j$ with each non-terminal R_j, $1 \leqslant j \leqslant m$, and k_j synthesised attributes a_l^S.

The semantic rules for each syntax rule in the grammar consist of assignments of the following forms or 'templates':

(I) $a_j^I(\langle R_l \rangle) := a_q^r(\langle R_p \rangle)$ where

(*a*) $1 \leqslant j \leqslant k_l$, $1 \leqslant q \leqslant k_p$
(*b*) p is the maximum possible index for which $0 \leqslant p \leqslant l \leqslant m$, such that the argument t_{lj} and t_{pq} represent the same variable.
(*c*) **if** $p = 0$ **then** $r = I$ **else** $r = S$.

(II) $a_j^S(\langle R_0 \rangle) := a_q^r(\langle R_p \rangle)$ where

(*a*) $1 \leqslant j \leqslant k_0$, $1 \leqslant q \leqslant k_p$
(*b*) p is the maximum possible index for which $0 \leqslant p \leqslant m$, such that the arguments t_{0j} and t_{pq} represent the same variable.
(*c*) **if** $p = 0$ **then** $r = I$ **else** $r = S$.

(III)
if $a_j^I(\langle R_0 \rangle) \neq c$ **and**
$a_j^I(\langle R_0 \rangle \neq$ **nil then**
$FLAG :=$ **false else** $a_j^S(\langle R_o \rangle) := c$

where $1 \leqslant j \leqslant k_0$ and t_{0j} is constant c.

(IV) $a_j^I(\langle R_p \rangle) := c$

where $1 \leqslant j \leqslant k_p$, $0 < p \leqslant m$ and t_{pj} is a constant c.

The data knowledge as explained in a previous section can be written in a Prolog-like notation as $R(c_1, c_2, \ldots, c_k)$ *is true*, where $c_1, c_2, \ldots c_k$ are constants. The corresponding attribute grammar rule is represented as:

$$\langle R \rangle ::= \textbf{nil}$$

The semantic rules are written for all c_j, $1 \leqslant j \leqslant k$ according to template III.

The attribute grammar rules, when the Prolog-like rules are known, can be generated mechanically from the corresponding templates. This task can also be automated with a preprocessor. Such a preprocessor can be incorporated in an ordinary attribute grammar interpreter to form an Extended Attribute Grammar interpreter similar to EAG of Ref. 4.

5. AN EXTENDED EXAMPLE

Let us now examine a more extended example.
The facts are:

the parent of Bob is John or *parent (John,Bob)* (7)
the parent of Liz is John or *parent (John,Liz)* (8)
the parent of Ann is Bob or *parent (Bob,Ann)* (9)
the parent of Pat is Bob or *parent (Bob,Pat)* (10)

The rules of the knowledge base are:

X is successor of Y if Y is parent of X or *succ(X,Y) if parent (Y,X)* (11)

X is successor of Y if Y is parent of Z and X is successor of Z or
succ(X,Y) if parent (Y,Z) and succ(X,Z) (12)

The questions are:

Who are the children of Bob? or *parent (Bob,?)* (13)
Who is predecessor of Pat? or *succ (Pat,?)* (14)

Each of the above sentences can be represented in attribute grammar notation as:

$\langle parent \rangle ::= \textbf{nil}$ (7)
if $a_1^I(\langle parent \rangle) \neq John$ **and** $a_1^I(\langle parent \rangle) \neq$ **nil**
then $FLAG :=$ **false else** $a_1^S(\langle parent \rangle) := John$;
if $a_2^I(\langle parent \rangle) \neq Bob$ **and** $a_2^I(\langle parent \rangle) \neq$ **nil**
then $FLAG :=$ **false else** $a_2^S(\langle parent \rangle) := Bob$;

(8), (9), (10), accordingly.

$\langle succ \rangle ::= \langle parent \rangle$ (11)
$a_2^I(\langle parent \rangle) := a_1^I(\langle succ \rangle)$;
$a_1^I(\langle parent \rangle) := a_2^I(\langle succ \rangle)$;
$a_1^S(\langle succ \rangle) := a_2^S(\langle parent \rangle)$;
$a_2^S(\langle succ \rangle) := a_1^S(\langle parent \rangle)$;

$\langle succ \rangle_0 ::= \langle parent \rangle \langle succ \rangle_1$ (12)
$a_1^I(\langle parent \rangle) := a_2^I(\langle succ \rangle_0)$;
$a_1^I(\langle succ \rangle_1) := a_1^I(\langle succ \rangle_0)$;
$a_2^I(\langle succ \rangle_1) := a_2^S(\langle parent \rangle)$;
$a_1^S(\langle succ \rangle_0) := a_1^S(\langle succ \rangle_1)$;
$a_2^S(\langle succ \rangle_0) := a_1^S(\langle parent \rangle)$;

$\langle question \rangle ::= \langle parent \rangle$ (13)
$a_1^I(\langle parent \rangle) := Bob$;
$output(a_2^S(\langle parent \rangle))$;

$\langle question \rangle ::= \langle succ \rangle$ (14)
$a_1^I(\langle succ \rangle) := Pat$;
$output\ (a_2^S(\langle succ \rangle))$;

In Figure 3*a,b* the two possible parse trees are shown for question (13) and in Figure 4*a,b* for question (14).

16-2

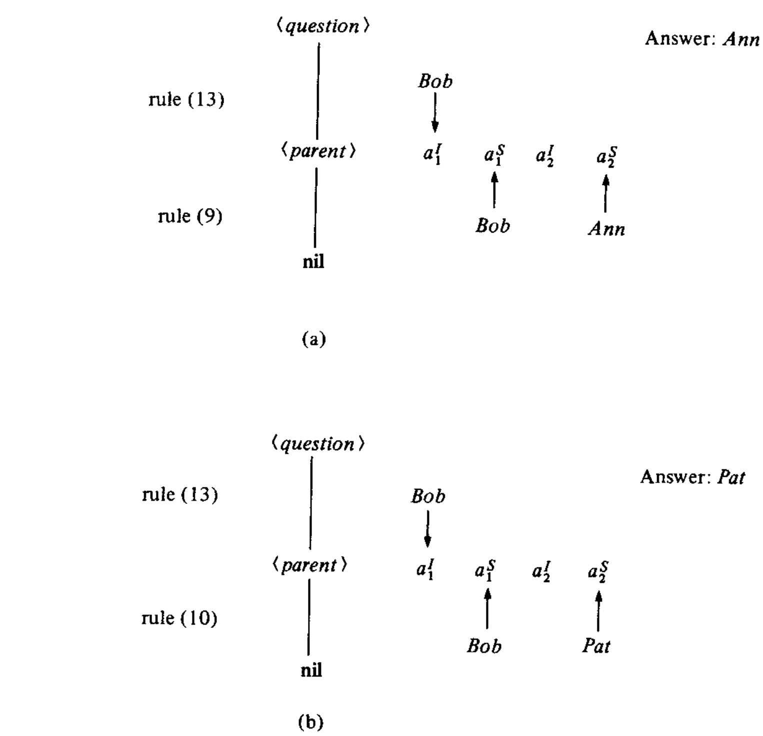

Figure 3. Parse trees of question (13).

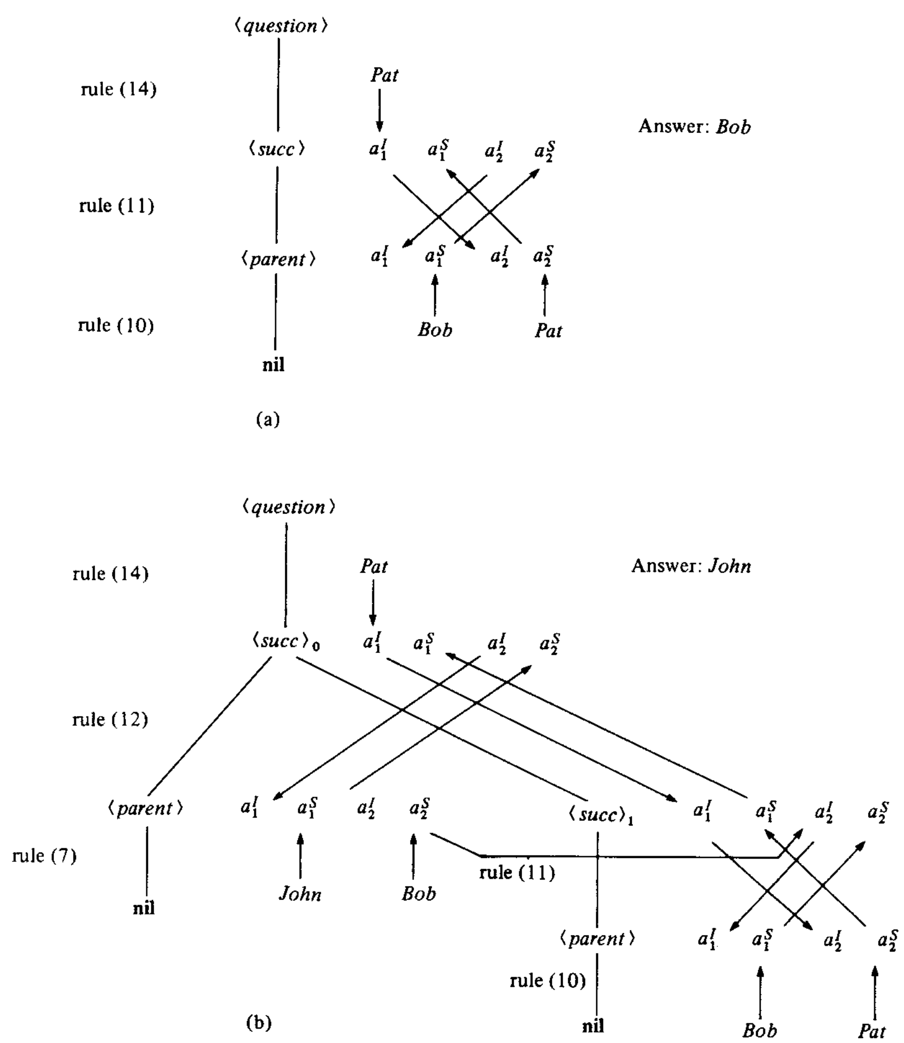

Figure 4. Parse trees of question (14).

5. CONTROL KNOWLEDGE REPRESENTATION

The interpretation of an attribute grammar is accomplished by an interpreter consisting of a parser and an attribute evaluator. The interpreter embodies the control knowledge necessary for activating the knowledge-base and deriving new facts from the data knowledge.

The parsing that takes place as already explained is degenerate, since it uses **nil** as its input string. The results are produced via attribute evaluation which is controlled by a logical metavariable *FLAG* that is set **true** or **false** in the semantics of the attribute grammar rules.

The attribute grammar interpreter of Ref. 6 was modified to use a more sophisticated backtracking mechanism,[14] in order to be able to obtain all possible solutions of a problem. This tool will be used for experimenting with knowledge engineering applications.

6. DISCUSSION

The feasibility of using attribute grammars for knowledge representation was examined in the present paper. There seems to be no obvious reason for not using attribute grammars for knowledge representation, and we might even argue that they may provide certain advantages. The first advantage that we consider stems from the fact that the software technology of attribute grammar processing is fairly mature. There exist many implementations of interpreters and compilers that can be used for applications.[15] The existence of compilers is extremely useful because compiled versions of grammars exhibit considerable efficiency of processing. Another advantage is the possibility offered to combine naturally declarative and procedural knowledge in a single tool. It is thus proposed that attribute grammar evaluators may prove to be useful knowledge engineering tools.

REFERENCES

1. D. E. Knuth, Semantics of context-free languages. In *Mathematical Systems Theory*, vol. 2, pp. 127–145 (1968).
2. R. E. Noonan, Structured programming and formal specification, *IEEE Trans. Software Eng.*, vol. SE-1, pp. 421–425 (Dec. 1975).
3. E. C. R. Hehner and B. A. Silverberg, Programming with grammars: an exercise in methodology-directed language design, *The Computer Journal* **26** (3), 277–281 (1983).
4. D. A. Watt and O. L. Madsen, Extended attribute grammars, *The Computer Journal* **26** (2), 142–153 (1983).
5. J. Kontos, Syntax-directed processing of texts with action semantics, *Cybernetica* **23** (2), 157–175 (1980).
6. G. Papakonstantinou, An interpreter of attribute grammars and its application to waveform analysis, *IEEE Trans. Software Eng.*, vol. SE-7 (3), pp. 279–283 (1981).
7. J. Kontos, Syntax-directed plan recognition with a microcomputer, *Microprocessing and Microprogramming* **9**(5), 277–279 (1982).
8. J. Kontos and G. Papakonstantinou, The interpretation of meta grammars describing syntax-directed interpreters using an attribute grammar interpreter, *IEEE Trans. Software Eng.*, vol. SE-8 (4), 435–436, (1982).
9. G. Papakonstantinou and F. Gritzali, Syntactic filtering of ECG waveforms, *Computers and Biomedical Research* **14**, 158–167 (1981).
10. G. Papakonstantinou, A sentence generator based on an attribute grammar, *Angewandte Informatik* **8** (1983).
11. G. Papakonstantinou and E. Skordalakis, Normal ECG pattern generation using an attribute grammar, *Proceedings of the 6th International Conference on Pattern Recognition.* IEEE Computer Society Press, Munich (1982).
12. D. S. Nau, Expert computer systems, *Computer*, **16**(2), 63–85 (1983).
13. G. V. Bochmann, Semantic evaluation from left to right, *CACM* **19**, 55–62 (1976).
14. R. W. Floyd, The syntax of programming languages – a survey, *IEEE Trans. Electronic Computers*, vol. EC-13 (4), 346–353 (1964).
15. K-J Räihä, Bibliography on attribute grammars, *SIGPLAN Notices* **15** (5), (1980).