Error Recovery with Attribute Grammars

A. Boccalatte, M. Di Manzo and D. Sciarra
Istituto di Elettrotecnica, Viale Francesco Causa 13, Viale Cambiaso 6, 16145 Genova, Italy

In this paper a parsing algorithm for attribute grammars is defined. This algorithm allows the detection and recovery from both syntactic and semantic errors, without introducing any delay in the parsing of correct sentences. Only synthesized attributes are considered. The aim of recovery procedure is only to overcome the error; no minimal distance correction is attempted.

INTRODUCTION

The problem of the semantic definition of a programming language has been approached by Knuth by means of 'attribute grammar'.[1] They consist of a syntactic part, which is typically a context-free grammar, and of a semantic part, made up of a set of attributes associated to each symbol and of a set of semantic functions used to evaluate the attributes' values while constructing the syntactic tree.

Attribute grammars have been widely investigated,[2–7] and rules have been given to evaluate attributes from left to right[3] and to build compilers which use semantic knowledge to overcome syntactic ambiguities.[4,6,7] A related concept is that of 'affix grammar', for which parsing methods have been defined.[8–10]

The main problem while parsing a sentence by means of an attribute grammar is to perform a suitable error detection and recovery. In Ref. 11 a parsing algorithm is presented, which allows the detection of a semantic error as soon as it is required to guarantee that the previously parsed substring of input symbols is a viable prefix.

A recovery procedure based on this method can operate only on the new input symbols, exactly as it happens with all the classical syntactic recovery procedures defined for LL(k) or LR(k) grammars.

Such a recovery algorithm is presented in the third section of this paper, after a short discussion of the theoretical preliminaries, which is included here in order to make the paper self-contained.

The last section is devoted to the detailed analysis of two examples of recovery from syntactic and semantic error.

THEORETICAL PRELIMINARIES

An attribute grammar consists of a context-free grammar $G_0(V_T, V_N, P, Z)$, where V_T is the set of terminal symbols, V_N is the set of non-terminal symbols, P is the set of production rules and $Z \in V_N$ is the distinguished symbol, and a set A_x of attributes associated with each symbol $x \in V_T \cup V_N$. Attributes can be divided into inherited attributes I_x and synthesized attributes S_x. A domain V_a is associated to each attribute a; V_a is the set of all the allowed values of a. A production $p \in P$ is written in the form:

$$p: x_0 \to x_1 x_2 \ldots x_{n_p}$$
$$x_0 \in V_N,\ x_i \in V_T \cup V_N \text{ for } 1 \le i \le n_p$$

If $a \in A_{x_i}$, the production p is said to have the *attribute occurrence* a, and this is written $a^{(i,p)}$. $A_{x_i}^{(i,p)}$ is the set of attribute occurrences associated with $a \in A_{x_i}$; $I_{x_i}^{(i,p)}$ and $S_{x_i}^{(i,p)}$ are defined similarly.

A domain $V_a^{(i,p)} = V_a$ is associated with every attribute occurrence $a^{(i,p)}$; therefore the set $A_{x_i}^{(i,p)}$ identifies a domain $N_{x_i}^{(i,p)}$ in a n-dimensional space where n is the number of elements of $A_{x_j}^{(i,p)}$.

In the same way $I_{x_i}^{(i,p)}$ identifies a domain $I_{x_i}^{(i,p)}$ and $S_{x_i}^{(i,p)}$ identifies a domain $S_{x_i}^{(i,p)}$.

The set of attribute occurrences:

$$M_p^0 = I_{x_0}^{(0,p)} \cup \bigcup_{i=1}^{n_p} A_{x_i}^{(i,p)}$$

identifies a domain M_p^0 in a m-dimensional space, where m_p is the number of elements of M_p^0. We may observe that M_p^0 represents the set of attribute occurrences whose values must be known in order to evaluate the synthesized occurrences of attributes associated with x_0.

The set of attribute occurrences

$$M_p^i = I_{x_0}^{(0,p)} \cup \bigcup_{j=1}^{i-1} A_{x_j}^{(i,p)} \quad 1 \le i \le n_p$$

identifies a domain M_p^i in a m-dimensional space, where m_i is the number of elements of M_p^i. M_p^i represents the set of attribute occurrences whose values must be known to evaluate the inherited attribute occurrences associated with x_i.

We can also define a set of functions $g_{(p,j)}^i$

$$M_p^i \to g_{(p,j)}^i \to S_{x_j}^{(j,p)} \quad 1 \le j \le i-1,\ 0 \le i \le n_p$$

A semantic function f_p^0 is associated with the set $S_{x_0}^{(0,p)}$ and a semantic function f_p^i is associated with each set $I_{x_i}^{(i,p)}$; these semantic functions determine the value of every attribute occurrence as a function of the values of other attribute occurrences in the same production.

The evaluation of these semantic functions depends on a set of semantic conditions C_p^i, which restrict the domain M_p^i to a valid domain $D_p^i \subset M_p^i$.

To obtain practical algorithms it is necessary to consider only synthesized attributes,[11] that is $I_x = \mathbf{0}$ for $\forall x: x \in V_T \cup V_N$.

Under this assumption, given an item

$$p: x_0 \to \alpha x_{j+1} \beta$$
$$\alpha, \beta \in V;\ \alpha = x_1 \ldots x_j;\ \beta = x_{j+2} \ldots x_{n_p}$$

we define: (1) *Valid residual domain or Residual domain* $D_{(p,j)}^0$ the domain

$$D_{(p,j)}^0 = D_{(p,0)}^0 \cap \bar{N}_{x_1}^{(1,p)} \ldots \bar{N}_{x_j}^{(j,p)}$$

CCC-0010-4620/82/0025-0331 $03.50

where $\bar{N}_{x_k}^{(k,p)}$ is a subset of M_p^0 characterized by the values of attribute occurrences $A_{x_k}^{(i,p)}$ for $1 \leq k \leq j$; $D_{(p,0)}^0 \subset D_p^0$ is the *initial residual domain*. (2) *Allowable domain* $N_{x_j}^{*(j,p)}$ the mapping of the function $g_{(p,j)}^0$ on the domain $D_{(p,j-1)}^0$; hence:

$$N_{x_j}^{*(p,j)} = g_{(p,j)}^0(D_{(p,j-1)}^0)$$

A more detailed discussion of the theoretical presuppositions can be found in Ref. 11.

ERRORS AND RECOVERY

The algorithm discussed in Ref. 11 allows the detection of a semantic error, extending to the handling of attributes the property of the viability of the prefix, which is typical of LL(k) and LR(k) parsers. However it doesn't suggest any recovery operation when an error is detected, and so a further insight in errors classification and management is needed. The classification of errors is trivial: we can have syntactic errors, due to a mismatching between the input symbol and the top of the stack, and semantic errors, caused by a set of input attributes (attributes of the new symbol in input) which is not included in the set of expected attributes. However, the recovery from a syntactic error, for which there is a number of well-known techniques,[12] has relevant semantic side-effects, that impose a global approach to the handling of both kinds of errors. The recovery from semantic errors is perhaps the simpler one.

Suppose that the parser is analysing the string $\alpha x \beta$, x being the current input symbol; suppose also that a production $p: y_0 \rightarrow y_1 y_2 \ldots y_i \ldots y_{n_p}$ exists, and let y_i be the current top stack.

A semantic error requires the fulfilment of the following two conditions: (1) $x \equiv y_i$ or x belongs to the directory set of y_i for a production p'; this condition will be written, in the following, as $x \in \sum_{y_i}^{p'}$.[2] (2) The actual values of the attributes of x do not match with the expected values. If $x \equiv y_i$ the mismatching can be formally described as

$$\mathrm{v}[\mathrm{A}_x] \not\subset N_{y_i}^{*(i,p)}.$$

To define the recovery action, we must remember that

$$V_a^{(i,p)} \equiv V_a, \quad \forall_a : a \in \mathrm{A}_{y_i}^{(i,p)}$$

This means that if a is an attribute of y_i, and hence of x too, the set of values of a allowed in the domain D_p^0 of the production p, and in all the residual domains $D_{(p,j)}^0$, $0 \leq j \leq i$, is a subset of V_a. Therefore it is always possible to change the set of actual values $\mathrm{v}[\mathrm{A}_x]$ into a new set $\mathrm{v}^*[\mathrm{A}_x]$, called recovery set, so that $\mathrm{v}^*[\mathrm{A}_x] \subset N_{y_i}^{\prime(i,p)}$. In most cases we will have a number of recovery sets $\mathrm{v}_1^*[\mathrm{A}_x] \ldots, \mathrm{v}_n^*[\mathrm{A}_x]$, being $\mathrm{v}_k^*[\mathrm{A}_x] \subset N_{y_i}^{*(i,p)}$, $1 \leq k \leq n$, and

$$\mathrm{v}_1^*[\mathrm{A}_x] \cup \mathrm{v}_2^*[\mathrm{A}_x] \cup \cdots \cup \mathrm{v}_n^*[\mathrm{A}_x] = N_{y_i}^{*(i,p)}$$

It is difficult to give a selection rule.

A semantic recovery action could cause the parser to detect new semantic errors, because it is actually a recovery operation and not a minimal distance correction. Hence the criterion could be to select the recovery set that minimizes the probability of causing new errors.

Unfortunately, it is impossible to foresee the consequences of one choice or another, and then the only practical suggestion is to select the recovery set which requires the minimum number of attributes values modifications.

As far as semantic errors are concerned, the case $x \in \sum_y^{p'}$ is very similar to the previous one; the mismatching is detected after one or more expansions of the syntactic tree, and can be recovered in the same way. However, some care must be taken because sometimes a semantic error exception can be raised by a purely syntactic error. Suppose, in fact, that two productions exist, p' and p'', which describe the non-terminal symbol y_i.

Suppose also that, within the current context, for semantic reasons, the symbol y_i must be expanded by means of production p'; this could mean that the expected values of the attributes of y_i cannot be synthesized using the production p''. If the new input symbol x satisfies the condition $x \in \sum_{y_i}^{p''}$, instead of the condition $x \in \sum_{y_i}^{p'}$, the production p'' will be used to expand y_i; but this syntactic operation will cause the allowable domain for x to be null, hence raising a semantic exception. However this error is not a true semantic error, because the correct recovery operation consists of deleting the symbol x or inserting a new symbol before it or changing it with a different symbol, and these are typical syntactic recovery actions. The recovery from a syntactic error is more complex, because of its semantic side-effects.

Let us apply a typical syntactic recovery procedure, as discussed in Refs 12 and 13, and consequently suppose that we are able to carry on the analysis with the new input symbol x' and the new top stack y_k. The allowable domain for x' must be evaluated, in order to check the attributes of the input symbol; this evaluation can be done only by supposing that we have actually found all the symbols which were over y_k in the stack and have been popped out.

Unfortunately, the knowledge about the values of the attributes of these symbols is not complete, because they have not been truly found, and therefore their actual values are unknown. Hence only their expected values can be used to evaluate the allowable domain of x'.

The procedure is as follows. Let D_{res} be the last evaluated residual domain before the discovery of the syntactic error, and let p be the current production.

Scan the symbols in the stack from y_i to y_k; if the last symbol of production p is found, evaluate $N^* = f_p^0(D_{\mathrm{res}})$, update p (p is now the new current production), pop the semantic stack and use N^* to calculate the new value of D_{res}. Repeat this evaluation for every symbol which terminates a production rule, until y_k is reached. The current D_{res} can now be used to calculate the allowable domain for x'.

A Pascal-like description of the parser and the recovery algorithm is now given.

Parsing algorithm

begin
/***Comment:** x = new symbol;
$\mathrm{v}[\mathrm{A}_x]$ = vector of attribute values of x;
y = top of the stack of symbols;
D_{res} = top of the stack of domains;
$\bar{p}$ = current production;
N_{amm} = current valid domain; */

```
while x ≠ ε do
  if x = y
    then if v[A_x] ∈ N_amm
      then
        begin
        while (x ≠ Z) (x is the rightmost symbol of p̄) do
          begin
          calculate N̄_x^(i,p̄); D_res ← D_res ∩ N̄_x^(i,p̄);
          x ← left part of p̄;
          v[A_x] ← f_p̄^0 (D_res);
          Pop the stack of domains and again define p̄
                                                and i;
          end;
        if x ≠ Z
          then
            begin
            calculate N̄_x^(i,p̄); D_res ← D_res ∩ N̄_x^(i,p̄);
            increment i by 1;
            N_amm ← g^0_(p̄,i) (D_res);
            syntactic action;
            end
          else syntactic action;
        end
      else semantic error;
    else if x ∈ Σ_y^p
      then if the right part of p is ≠ ε
        then
          if N_amm ∩ f_p^0 (D_p^0) = 0
            then syntactic error
            else
              begin
              N_amm ← N_amm ∩ f_p^0(D_p^0); D^0_(p,0) ←
                                        f_p^0(-1)(N_amm);
              push D^0_(p,0) into the stack of domains;
              p̄ ← p; i ← 1;
              N_amm ← g^0_(p̄,i) (D_res);
              syntactic action;
              end
        else
          if 'null' ∈ N_amm for all attributes of y
            then
              begin
              insert y before x into the input string;
              x ← y; v[A_x] ← 'null';
              end
            else syntactic error
      else syntactic error;
end
```

Initially the symbols stack contains Z and $N_{amm} = N_z^{(0,i)}$. The occurrence of a syntactic error causes a call to the recovery procedure described below.

Syntactic recovery algorithm

```
begin
  y' ← y;
  while (x ≠ y') and (x ∈ Σ_y'^p) do
    begin
      if y' is not the last symbol in the stack of symbols
        then y' ← symbol following y' in the stack
        else
          begin
          y' ← y; x ← symbol following x in the input
                                                string
          end
      end
    while y' ≠ y do
      begin
        if y is the rightmost symbol of p̄
          then
            begin
            z ← left part of production p̄;
            v[A_z] := f_p^0 (D_res);
            pop the stack of domains and again define
                                                p̄ and i;
            calculate N̄_z^(i,p̄); D_res ← D_res ∩ N̄_z^(i,p̄);
            i ← i + 1;
            end
          else i ← i + 1;
        pop the stack of symbols;
      end
  N_amm ← g^0_(p̄,i)(D_res);
end.
```

A semantic error detection results in the following simple recovery procedure.

Semantic recovery algorithm

```
begin
  look for the item of N_amm for which the distance from
  v[A_x] is minimal; let a be such an item;
  v[A_x] ← a;
end.
```

EXAMPLE OF RECOVERY

We now discuss an example of recovery from syntactic and semantic errors. This example is quite long because it is not easy to find a case both simple and significant.

Let us consider the grammar of Fig. 1, which describes simple declarative or imperative sentences. The directory set is associated to each production.

The attributes of terminal and non-terminal symbols are shown in Fig. 2, while in Fig. 3 the domains associated to each production are described. If we now try to analyse the sentence 'Now the little boy ate an apple', when entering the symbol 'ate' a semantic error is detected, because the value of its tense attribute does not match with the value of the attribute of 'now'. This error is detected after the eleventh step of the parsing procedure, in fact the input symbol 'Verb1' (see Fig. 5) has the attribute values '2, A, T' and this triple is not contained in N_{amm}.

Figure 4 shows the current production number, the index of the current symbol within the current production, the domain, the current content of the stacks of domains and symbols, the input symbol and the fraction of the sentence that must still be parsed when the error is detected. The admissible triple nearest to the current values of the attributes of Verb1 is '1, A, T' and so the parsing procedure is resumed after having assigned this triple to Verb1.

The case of syntactic error is more complex. Let us consider the sentence 'now the little an apple' where the words 'boy' and 'eats' are omitted. The syntactic error is discovered after the tenth step of the parsing procedure shown in Fig. 5, where the recovery algorithm is entered

1. ⟨F⟩ : : = DEC #	**Pref1, Art1, Adj1, Noun**
2. ⟨F⟩ : : = IMP #	**Don't, Verbinf**
3. ⟨IMP⟩ : : = ⟨AUX1⟩ **Verbinf** ⟨OBJ⟩	**Don't, Verbinf**
4. ⟨DEC⟩ : : = ⟨PREF⟩⟨DEC1⟩	**Pref1, Art1, Adj1, Noun**
5. ⟨DEC1⟩ : : = ⟨NG⟩⟨VERB⟩⟨OBJ⟩	**Art1, Adj1, Noun**
6. ⟨NG⟩ : : = ⟨ART⟩⟨ADJ⟩ **Noun**	**Art1, Adj1, Noun**
7. ⟨NG⟩ : : = **Pron**	**Pron**
8. ⟨OBJ⟩ : : = ⟨NG⟩	**Art1, Adj1, Noun**
9. ⟨OBJ⟩ : : = ε	#
10. ⟨VERB⟩ : : = **Verb1**	**Verb1**
11. ⟨VERB⟩ : : = **Aux Verbinf**	**Aux**
12. ⟨ART⟩ : : = **Art1**	**Art1**
13. ⟨ART⟩ : : = ε	**Adj1, Noun**
14. ⟨ADJ⟩ : : = **Adj1**	**Adj1**
15. ⟨ADJ⟩ : : = ε	**Noun**
16. ⟨PREF⟩ : : = **Pref1**	**Pref1**
17. ⟨PREF⟩ : : = ε	**Art1, Adj1, Noun**
18. ⟨AUX1⟩ : : = **Don't**	**Don't**
19. ⟨AUX1⟩ : : = ε	**Verbinf**

Figure 1

F	1^0	D1 (present declarative) D2 (past declarative) IM (positive imperative) IMN (negative imperative)
IMP	1^0	IM (positive imperative) IMN (negative imperative)
DEC	1^0	1 (present tense) 2 (past tense)
DEC1	1^0	1 (present tense) 2 (past tense)
NG	1^0	A (living) O (non-living)
OBJ	1^0	E (OBJ exists) N (OBJ does not exist)
VERB	1^0	1 (present tense) 2 (past tense)
	2^0	A (verb implying living subject) O (verb implying general subject)
	3^0	T (transitive) I (intransitive)
ART	1^0	S (singular article) P (plural article) N (ART does not exist)
ADJ	1^0	A (referred to living entity) O (referred to general entity) N (ADJ does not exist)
PREF	1^0	1 (present tense) 2 (past tense) N (PREF does not exist)
AUX1	1^0	E (AUX1 exists) N (AUX1 does not exist)
Verbinf	1^0	A (verb implying living subject) O (verb implying general subject)
	2^0	T (transitive) I (intransitive)
Noun	1^0	S (singular) P (plural)
	2^0	A (living entity) O (non living entity)

Figure 2. Attributes of terminal and non-terminal symbols.

Production 1

V(F)	V(DEC)	V(#)
D1	1	E
D2	2	E

Production 2

V(F)	V(IMP)	V(#)
IM	IM	E
IMN	IMN	E

Production 3

V(IMP)	V(AUX1)	V(Verbinf)	V(OBJ)
IM	N	AT	E, N
IM	N	OT	E, N
IM	N	AI	N
IM	N	OI	N
IMN	E	AT	E, N
IMN	E	OT	E, N
IMN	E	AI	N
IMN	E	OI	N

Production 4

V(DEC)	V(PREF)	V(DEC1)
1	1	1
2	2	2
1	N	1
2	N	2

Production 5

V(DEC1)	V(NG)	V(VERB)	V(OBJ)
1	A	1AT	E, N
1	A	1AI	N
1	A	1OT	E, N
1	O	1OT	E, N
1	A	1OI	N
1	O	1OI	N
2	A	2AT	E, N
2	A	2AI	N
2	A	2OT	E, N
2	O	2OT	E, N
2	A	2OI	N
2	O	2OI	N

Figure 3. Domains associated to each production.

Production 6

V(NG)	V(ART)	V(ADJ)	V(Noun)
A	S	A	SA
A	S	O, N	SA
O	S	O, N	SO
A	P	A	PA
A	P	O, N	PA
O	P	O, N	PO
A	N	A	SA
A	N	A	PA
A	N	O, N	SA
A	N	O, N	PA
O	N	O, N	SO
O	N	O, N	PO

Production 7

V(NG)	V(Pron)
A	A

Production 8

V(OBJ)	V(NG)
E	A, O

Production 9

V(OBJ) = N

Production 10

V(VERB)	V(Verb1)
1AT	1AT
1AI	1AI
1OT	1OT
1OI	1OI
2AT	2AT
2AI	2AI
2OT	2OT
2OI	2OI

Production 11

V(VERB)	V(Aux)	V(Verbinf)
1AT	1	AT
1AI	1	AI
1OT	1	OT
1OI	1	OI
2AT	2	AT
2AI	2	AI
2OT	2	OT
2OI	2	OI

Production 12

V(ART)	V(Art1)
S	S
P	P

Production 13

V(ART) = N

Production 14

V(ADJ)	V(Adj1)
A	A
O	O

Production 15

V(ADJ) = N

Production 16

V(PREF)	V(Pref1)
1	1
2	2

Production 17

V(PREF) = N

Production 18

V(AUX1)	V(Don't)
E	E

Production 19

V(AUX1) = N

Figure 3 (continued)

with x = Art1(S), y = Noun, $\bar{p} = 6$, $i = 3$ and the following steps are performed (initially x = Art1, y = Noun).

1. $y' \leftarrow$ Noun;
2. $y' \neq x$ and $x \notin \sum_{y'}^{p}$ and y' is not the last symbol of the stack, then $y' \leftarrow$ VERB
3. $y' \neq x$ and $x \notin \sum_{y'}^{p}$ and y' is not the last symbol of the stack, then $y' \leftarrow$ OBJ
4. $x \in \sum_{y'}^{8}$, $y' \neq y$ and y' is the rightmost symbol of production $\bar{p}$, then do the following: $z \leftarrow$ NG; $v[A_z] \leftarrow [{}^{A}_{0}]$; pop the stack of domains; $\bar{p} \leftarrow 5$; $i \leftarrow 1$; $\bar{N}_z^{(1,5)} \leftarrow D_5^0$; $D_{res} \leftarrow [\gamma]$ (γ is defined as in Fig. 4); $i \leftarrow 2$; pop the stack of symbols (now y = VERB)
5. $y' \neq y$ and y' is not the rightmost symbol of the production $\bar{p}$, then do: $i \leftarrow 3$; pop the stack of symbols (now y = OBJ)
6. $y' = y$, then $N_{amm} \leftarrow [{}^{E}_{N}]$ and exit from the recovery algorithm.

The parsing procedure can now be resumed starting from the 14th step.

CONCLUSIONS

The parsing algorithm discussed in this paper allows the handling of both syntactic and semantic errors in an integrated manner.

With this technique no backtracking is needed when an error is detected, so the typical properties of the LL(k) syntactic parsers are extended to the semantic analysis.

The most important drawback of the proposed method is its inability to handle inherited attributes. A deterministic parsing with inherited attributes is still an open problem, because the values of the attributes of a symbol depend on the values of the attributes of the symbols which have been already parsed; therefore it is not trivial to define a method to look ahead for possible errors while evaluating this kind of attributes. The work in this field is still in progress.

$$[\beta] = \begin{bmatrix} 1 & E \\ 2 & E \end{bmatrix} = D_1^0 \qquad [\theta] = \begin{bmatrix} S & A & SA \\ S & O,N & SA \\ S & O,N & SO \end{bmatrix} \qquad [\eta] = \begin{bmatrix} A & 1AT & E,N \\ A & 1AI & N \\ A & 1OT & E,N \\ A & 1OI & N \end{bmatrix}$$

$$[\tau] = \begin{bmatrix} 1,N & 1 \\ 2,N & 2 \end{bmatrix} = D_2^0 \qquad [\pi] = \begin{bmatrix} S & N & SA \\ S & N & SO \end{bmatrix} \qquad [\mu] = \begin{bmatrix} 1AT \\ 1AI \\ 1OT \\ 1OI \end{bmatrix}$$

$$[\gamma] = \begin{bmatrix} A & 1AT & E,N \\ A & 1AI & N \\ A,O & 1OT & E,N \\ A,O & 1OI & N \end{bmatrix} \qquad [\varepsilon] = \begin{bmatrix} S & O & SA \\ S & O & SO \end{bmatrix} \qquad [\alpha] = [A \quad 1AT \quad E,N]$$

p	i	N_{amm}	D_{res} stack	Symbol stack	x	Input string
6	3	SA, SO	≠ (1, 0, (β)) (4, 1, (1, 1)) (5, 0, (γ)) (6, 2, (ε))	≠ 1 # OBJ VERB Noun	Verb1(2AT)	Art1(S) Noun(SO) # (E)
5	2	1AT, 1AI, 1OT, 1OI	≠ (1, 0, (β)) (4, 1, (1 1)) (5, 1, (η))	≠ 1 # OBJ VERB		
10	1	1AT, 1AI, 1OT, 1OI	≠ (1, 0, (β)) (4, 1, (1 1)) (5, 1, (η)) (10, 0, (μ))	≠ 1 ≠ OBJ Verb1		

Error: $V(A_x) = 2AT \notin N_{amm}$; $N_{amm} = \begin{bmatrix} 1AT \\ 1AI \\ 1OT \\ 1OI \end{bmatrix}$

Figure 4. Situation after the 11th step of the parsing procedure (semantic error detected).

p	i	N_{amm}	D_{res} stack	Symbol stack	x	Input string
14	1	A, O	≠ (1, 0, (β)) (4, 1, (1 1)) (5, 0, (γ)) (6, 1, (θ)) (14, 0, (A or O))	≠ 1 # OBJ VERB Noun 14 Adj1	Art1(S)	Noun(SO) # (E)
6	3	SA, SO	≠ (1, 0, (β)) (4, 1, (1 1)) (5, 0, (γ)) (6, 2 (ε))	≠ 1 # OBJ VERB Noun		

$$N_{amm} = \begin{bmatrix} SA \\ SO \end{bmatrix} \qquad D_{res} = \begin{bmatrix} S & O & SA \\ S & O & SO \end{bmatrix}$$

Figure 5. Situation after the tenth step of the parsing procedure (syntactic error detected).

REFERENCES

1. D. E. Knuth, Semantics of context-free languages, *Math. Systems* Th.2, 127–145 (1968).
2. G. V. Bochmann, *Semantic Equivalence of Syntactically Related Attribute Grammars*. Publ. 148, Department d'Informatique, Montreal (1969).
3. G. V. Bochmann, *Semantic evaluation from left to right, Communications of the ACM* **19**, 55–62 (1976).
4. G. V. Bochmann and P. Ward, *Compiler writing system for attribute grammars. The Computer Journal* **21** (No. 2), 144–148 (1978).
5. K. Culik, *Attributable Grammars and Languages*, Publ. 3, Department d'Informatique, Montreal (1969).
6. D. R. Milton, L. W. Kirchhoff and B. R. Rowland, An ALL(1) compiler generator. Proceedings of SIGPLAN symposium on Comp. Constr., *ACM SIGPLAN Notices* **14** (No. 8), 152–157 (1979).
7. M. Saarinen, On constructing efficient evaluators for attribute grammars, in *Automata, Languages and Programming, Fifth Colloquium*, ed. by G. Ausiello and C. Böhm, pp. 382–397. Springer-Verlag, Heidelberg (1978).
8. C. H. A. Koster, Affix grammars, in *ALGOL 68 Implementation*, p. 95. North-Holland, Amsterdam (1971).
9. C. H. A. Koster, *Two level Grammars*, Seminar on Automata and Mathematical Linguistic, Matematsch inst., Budapest (May 1970).
10. D. Crowe, Generating parsers for affix grammars. *Communications of the ACM* **15**, 728–732 (1972).
11. A. Boccalatte and M. Di Manzo, An approach to the detection of semantic errors. *The Computer Journal* **23** (No. 4), 317–323 (1980).
12. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts (1977).
13. S. Crespi-Reghizzi, P. Della Vigna and C. Ghezzi, Trattamento degli errori semantici. *Atti del Congresso AICA, Genova*, 297–307 (1975).
14. G. Lyon, Syntax-directed least error analysis for context-free languages—a practical approach. *Communications of the ACM* **17**, 3–14 (1970).

15. D. J. Rosenkrantz and R. E. Stearns, Properties of deterministic top-down grammars. *Information and Control* **17**, 226–256 (1970).
16. A. Van Wijngaarden, *Orthogonal design and description of a formal language*, Mathematisch Centrum Report MR 76, Amsterdam (1965).
17. W. T. Wilner, Declarative semantic definition, Rep. STAN-CS-233-71, Computer Science Dept., Stanford (1971).
18. W. T. Wilner, Formal semantic definition using synthesized and inherited attributes, in *Formal Semantics of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey (1972).

Received February 1981

Book Reviews

Continued from p. 396

R. S. HAYES AND C. R. BAKER
Simplified Accounting for the Computer Industry
Wiley, Chichester, 1981. 191 pp. £15.95.

The title of this book suggests a textbook capable of educating computer personnel only in the art of accounting principles. It certainly achieves this objective and indeed it also provides a valuable insight for the accountant on how the computer professional will design and implement a computer system. There is a great deal of information describing how the system designer goes about his job or how he should do so. How often we find the analyst to be a raw recruit learning his trade at the expense of others. The method of presenting the information is a refreshingly new approach to loading the human computer with facts. The book starts like any Ian Fleming novel. 'You could feel the sun, but it was not hot. The sky was the blue of lapis lazuli. Kimberly Rogers sat in the garden looking at its reflection in a pool of water . . .'. And if that is not enough to get any male accountant or systems analyst interested, it turns out that Kimberley is attractive, sophisticated and black and she has a passion . . . for knowledge. The story centres around the implementation of a computerized accounting system for Hero Manufacturing by Kimberly and her Egyptian boyfriend who is an accountant. Chapter by chapter she tells us how she approaches each assignment and then provides accounting details and principles of implementing such a system. It is interesting to note that the implied hardware is a mini or micro on-line system and not the expensive monster of the past. In order to provide the necessary accounting principles to back-up the computerized systems, Kimberly's boyfriend gives a series of lectures which are well-presented and easy to follow, and like any good lecturer allows time for questions at the end and a summary. The book explains the accounting terms to the computer man and computer terms to the accountant in a manner that will make the reader want to go on the next chapter like the storyteller in her search for knowledge. I believe this book will be invaluable to systems analysts and accountants who wish to know more about their brothers' skills.

T. M. BARNARD
London

A. S. TANENBAUM
Computer Networks
Prentice-Hall, Englewood Cliffs, New Jersey, 1981. 517 pp. £18.20.

There is an ambiguity in the title of this book—is it a book about networks *for* computers, i.e. a book about data transmission networks? Or is it a book about networks *of* computers, that is a book about a collection of linked machines forming a distributed system? One suspects that the ambiguity may be deliberate. In any event, the book treats both aspects, although the principal emphasis is on data transmission networks, realized of course as a network of computers. The later chapters touch on the systems aspects of distributed systems, but cannot be regarded as a full treatment of the subject.

The main body of the material is a thorough treatment of the current situation on data transmission, based on the notions of the 150 'seven layer' system. This naturally, and properly, forces a highly structured approach to the subject, and the author uses this structure to map three major networking systems, the ARPA net, IBM's SNA, and Digital's DECNET on to the 150 model. As is cheerfully admitted, this mapping is at some points a little strained. Within this framework, the emphasis is on wide area networks, operating over noisy lines of limited bandwidth and high recurrent cost, but again with one chapter dealing with the rather different problems encountered for local area networks, especially those using Aloha-type broadcasting. For UK readers, the rather cursory treatment of ring-based systems may be rather disappointing.

Overall this book presents a well-balanced treatment of its subject area—I enjoyed reading it, and hope that my own students will find it as enjoyable when I recommend it to them.

M. WELLS
Leeds

GIJSBERT VAN DER LINDEN (ED.)
APL80
North-Holland, Amsterdam, 1980. 370 pp. $48.75.

This book is a 'must' for the devotees of APL. It consists of papers presented at an international conference on APL held in June 1980 in the Netherlands. After an introductory paper by Iverson himself the remaining papers are grouped under the headings: Use of language, Methodology, Simulation, APL systems, Programming techniques, Design of language, Application, Database applications, Education, Data analysis and Special topics and also included are a handful of papers by invited speakers.

The reviewer has always been suspicious of APL enthusiasts about the use of APL in business computing, largely on the grounds that a key element in business data processing is the detection and handling of input errors, a topic which few APL specialists have seemed to find interesting. It was therefore salutary to find two papers (Mayforth on 'APL as a language for applications programmers' and Gardner and Swain on 'A group of input utility functions') specifically addressing this problem. My guess is that almost everyone with a knowledge of APL will find something of interest in this book.

P. G. RAYMONT
Manchester

PATRICK D. T. O'CONNOR
Practical Reliability Engineering
Heyden, London, 1981. 300 pp. £12.00.

The area of the many aspects of reliability is hazy for many computer professionals and yet is one of which we are all going to become increasingly aware, as systems are used in ever-increasing numbers.

To those professionals who have not yet considered reliability or do not know where to begin, I can readily recommend this book due to its very easy style, step-by-step introduction and comprehensive coverage. The many references to published works also serve to aid those who, having read the book, wish to learn more.

Before anyone should reject the book on the grounds that it is concerned with 'engineering', I would like to refer them to Chapter 8, 'Software Reliability', which forms an extremely good starting point for the computer professional.

Overall the book is of a high quality, good presentation and contains few errors.

P. A. BENNETT
Brigg