Methods	Necessary Time in Seconds	
	Serial	Parallel
(2)	39	25
(6) and (7)	39	25
(13) and (14)	41	26

We have also compared the results we obtained with the analytical results coming out of (17), for a few points in the domain at issue :

t	Method	$x=0,3$ or $x=0,7$		$x=0,5$	
		Numerical Solution	Analytical Solution	Numerical Solution	Analytical Solution
0,1	(2)	0.24723	0.24441	0.30562	0.30211
	(6) and (7)	0.25027		0.30842	
	(13) and (14)	0.24909		0.30708	
0,3	(2)	0.033895	0.033953	0.041897	0.041968
	(6) and (7)	0.036638		0.045144	
	(13) and (14)	0.036197		0.044618	
0,5	(2)	0.0047393	0.0047165	0.0058581	0.00583
	(6) and (7)	0.0053632		0.0066085	
	(13) and (14)	0.0052597		0.0064834	
0,7	(2)	0.00066266	0.0006552	0.00081909	0.00080986
	(6) and (7)	0.0007851		0.00096738	
	(13) and (14)	0.00076428		0.000942	
0,9	(2)	0.000092654	0.000091016	0.00011453	0.0001125
	(6) and (7)	0.00011493		0.00014161	
	(13) and (14)	0.00011213		0.00013822	

We have chosen $x=0,5$ mostly because $f(x)$ is not derivable in this point, and $x=0,3$ or $x=0,7$ so as to draw a comparison between the approximation in these points and the points with $x=0,5$.

REFERENCES

1. C h i r u ţ ă, C. — *Metode numerice de rezolvare a ecuaţiilor diferenţiale cu derivate parţiale adaptate calculului paralel.* Paper held at the 6th Edition of the Informatics Colloquim INFO—IAŞI '87, October, 9th—10th, 1987, p. 146—152.
2. E v a n s D. J. — *Parallel Algorithms in Computational Linear Algebra.* Parallel Computer and Computation, C.W.I. Newsletter, nr. 9, 1985, p. 55—79.
3. I x a r u, Gr. L. — *Metode numerice pentru ecuaţii diferenţiale cu aplicaţii.* Editura Academiei R.S.R., 1979, 208 p.
4. K o t o v, V. E. — *Elementî parallelinovo programmirovania.* Radio i sviazi, 1983, 240 p.

Received 9.IV.88

Faculty of Mathematics
University „Al. I. Cuza" Iaşi,
Iaşi R-6600, Romania

ANALELE ŞTIINŢIFICE ALE UNIVERSITĂŢII „AL. I. CUZA"
Tomul XXXIV, s. I a. Matematică-Informatică, 1988, f. 4

SYNTAX ORIENTED DATA PROCESSING

BY

LUCA DAN ŞERBANAŢI, VALENTIN CRISTEA, ADRIAN POREMBSCHI, MIHAELA SABIN, DANIEL SABIN, ADRIAN PRIVACHE

Introduction. Developing a program, the programmer aims for it to accomplish a certain computation. By writing code, he assembles various computer operations. In other words, while being preoccupied by syntax, the programmer must constantly think in terms of semantics. By being fully aware of the difference between a symbol and its meaning, between an operator and an operation, between a program and a processor which executes it, or in general terms, between syntax and semantics, the programmer has in mind, explicitly or not, the relations between the syntactical structures of a program and the semantical ones.

As far back as 1961, I r o n s suggested in [1] the existence of such a relation in the compilation process. Today, a widely accepted idea is that access to semantics is accomplished exclusively by way of syntax. This being the case we can use it in the design and implementation of many data processings.

This paper sets out to classify the most frequently encountered syntax oriented data processings, to introduce models for these processings, as well as solutions for their implementation.

1. Syntax and semantics. We shall consider, in the following, that the main formal operational model in the study of the programming languages syntax continues to be the context-free grammar (*CFG*). In what concerns semantics, no such formal model, similar to the one mentioned above, exists, at least not in terms of suggestiveness, conciseness or ease in handling, although several such proposals for models have been made ([2], [3], [4], etc.).

Irrespective of the formal model chosen for the semantics, one can easily ascertain that the syntax of a context-free language (*CFL*) is allways richer in information than any semantics of the language and any such associated semantics actually makes the language "poorer" in terms of diversity of expression. For example, modifying the effect of a program (in other words its meaning) can be obtained exclusively by modifying the program text, thus its syntactical structures. In contrast, different syntactical programs (i.e. with nonidentical texts) can have identical effects. In formal terms, if L is a language (i.e. a subset of the set of symbols over an alphabet) and D the domain of sentence evaluation, a language semantics is an application : $L \rightarrow D$.

From *ALGOL-60* onwards, any programming language uses *CFG* for syntactical definition and its semantics is introduced related to its syntactical components. That is, from the definition mode of the semantics itself, the syntactical structuring operators induce a structure in *D*.

The *ADJ* group has expressed this fact in a concise and general manner in [2] with the aid of multi-sorted operator algebras.

"Syntax is an initial algebra S in a class of algebras, and any other algebra A in the class is a possible *semantic domain* (or semantic algebra); the *semantic function* is the uniquely determined homomorphism $h_A: S \to A$, assigning a meaning $h_A(s)$ in A to to each syntactic structure s in S".

In the particular case of context free languages (assimilable, up to a point, by programming languages), the structuring operators are exactly the grammar productions which define the language syntax. The productions structure not only the language sentences, but their meaning. Comprising or modifying a syntactical structure from the programming language that we are working with, we create new meanings; this is the most direct mode we have to "build" or "rebuild" computations with the purpose of their execution by a computer.

2. Concrete syntax and abstract syntax. As far back as 1962 McCarthy, in [2], signals the existence of two kinds of syntax: *concrete syntax*, well known to any user of a programming language, because it is the one presented in the language definition manual, and *abstract syntax*, an improved variant of the first, independent of the notations used in the program text, reflecting only the essential structural-algebric properties of the language sentences. The main characteristic of abstract syntax is that it uses syntactical components bearing semantical value, exclusively. These are, as a rule, the grammar nonterminals which makes abstract syntax use only nonterminals. In order to achieve the desired aim, that of describing the essential structural properties, the use of only nonterminals in the grammar productions is considered sufficient.

The distinction between the two syntaxes becomes clearer if we imagine two different scenarios, dealing with the writing of a program, starting from the two above mentioned syntaxes.

The first scenario is the one known, in which the succession in the program conception is:

text→syntactical structure→accomplished computation

In other words, the projection of a program into computation is accomplished in two steps: from the text, one raises (mentally, of course) the syntactical structure, and from it the necessary com putation is therefore deduced. If it differs in any way from the desired one, the entire cycle, formed by the two steps, is resumed. The second scenario is used more seldomly. The sequence of steps in conceiving the program takes the form:

$$\text{syntactical structure} \begin{cases} \longrightarrow \text{text} \\ \longrightarrow \text{accomplished computation} \end{cases}$$

One notes a direct passing, in a single step, from the working form, which is the syntactical structure, to the accomplished computation, it being the semantical evaluation of the structure. The transition towards text, with the aim of execution, becomes optional when we have at our disposal an evaluator of syntactical structures in the respective semantical domain, in the same way that, by way of language processors, we have at our disposal — in the first scenario — text evaluators. If the evaluator is missing, the transition towards text takes place once, at the end of the program conception process. The transition can also take place automatically. the programmer taking advantage at the same time, both of the structure and of the text, if he has at his disposal a parser from the abstract syntax to the concrete one. At last, work on the syntactical structure is much essened by the use of a syntax oriented editor.

In the following, all references to the term syntax will relate to abstract syntax. As in the case of concrete syntax, the most natural representation of the structures of abstract syntax is the tree-like one.

3. From schema to model. The homomorphism $h: S \to A$ is a simple schema representing the relation syntax-semantics. But any schema proves its usefullness only when it can be extended into a sufficiently elaborate model, in order to allow the implementation to be systematized.

In order to achieve a model acceptable to the implementation, one should know about the fashion in which syntax is embedded into the semantical domain, in other words, how to build the algebra A starting from S. This construction usually takes place in two steps:

a) The so-called „carriers" are established, in other words the sets of values for each sort. In the case of languages, the sets associated to each nonterminal are established.

b) A function associated to each operator from the initial algebra S is created. The function is defined over a domain encompassing carriers of a certain sort taking as their model the manner in which the operator is defined in the S algebra. In the case of the initial algebra G (a *CFG*) of a language, to each production:

$$i: X_0 \to X_1 X_2 \ldots X_k$$

one associates a function:

$$p_i: M(X_1) \times M(X_2) \times \ldots \times M(X_k) \to M(X_0).$$

in which $M(X_j)$ is the carrier for the nonterminal X_j.

The nature of the carriers can represent a classifying criterion for the models used in syntax directed processings. Without exhausting all the possible cases, we shall present three of the most commonly encountered ones. In real cases, they appear most often in a combined manner. The three cases have, as one can easily ascertain, different degrees of generality.

A) The most simple case is that of carriers formed of sets of symbol strings (characters, in particular): to each nonterminal we associate such a set, and the semantics will describe an evaluation of syntactical structures into symbol strigns. In particular, an important case is that of the isomorphic algebra with G, the algebra of preorder linearized syntactical trees.

B) The carriers associated to nonterminals can differ from one nonterminal to another, or in general, the carriers can have nonhomogeneous structures (cartesian sets of different kinds).

C) The carriers are sets of actions. In view of defining them, we shall introduce the following terms: the *informational context* (a data structure) and the set of *action primitives* (procedures, functions etc., working on the context). These last ones have a double role, both that of context processings (initialization, inspection, modification) as well as that of primitives enchaining (with sequence, decision, selection and cycle operations). In this case, the semantic field can be that of context processing programs, i.e. programs which are represented by primitive sequences (in the compilative variant), or when we force the primitive execution, it can be composed of the set of context values (in the interpretive variant), being sets of such sequences, the semantics will actually describe their composition. An example of the usefullness of such semantics is the possibility of expressing eventual context dependencies of the language.

One can look upon the three cases as being translations of the syntactical structures into values of the semantic field.

The models for the three cases are formal systems which are based on a *CFG*. Enriching the formalism provided by the *BNF* (B a c k u s N a u r Form) with new metasymbols having interpretations specific to the desired effect: that of achieving syntax oriented processing we obtain, in effect, description languages for the syntax-semantics relation. Due to their suggestiveness, the respective languages can imply solutions for the automation of the semantical evaluation.

4. Case A. In figure 1a) is presented the abstract syntax of a language, whose concrete representation is shown in figure 1b) as being an evaluation process in the set $\{(\,,)\}^*$. (The notation A^* is used to describe the operation "star" upon the set A).

Example 1

a)	b)
1. $<S> ::= <S> <S>$	1. $<S> ::= \{(\} <S> \{)\} <S>$
2. $<S> ::=$	2. $<S> ::=$

Figure 1

The metasymbols {, } delimit strings of characters from the semantics field. A derivation in the formal system presented in 1b) appears as follows:

$$<S> \Rightarrow \{(\}<S>\{)\}<S> \Rightarrow \{(\}\{(\}<S>\{)\}<S>\{)\}<S> \Rightarrow$$
$$\Rightarrow \{((\}<S>\{)\}\{)\}<S> \Rightarrow \{((\}<S>\{))\}\{(\}<S>\{)\}<S> \Rightarrow$$
$$\Rightarrow \{((\}\{))\,(\}<S>\{(\}<S> \Rightarrow \{(())\,()\}<S> \Rightarrow \{(())()\}.$$

The possibility of concatenating strings delimitated by {, } becomes apparent when the groups are adjacent. The evaluation present in the derivation is that of the syntactical structure in Figure 2, in the string "(()) ()".

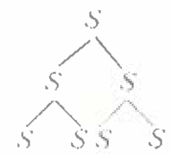

Figure 2

Generally speaking, syntactical structures evaluations described by the grammar in Figure 1a) lead to correctly balanced strings of paranthesis. On the other hand, the grammar shown in figure 3 can be linked to the concrete syntax of a language whose abstract syntax is presented in Figure 1a).

$$1.\ <S> ::= (\ <S>\ )\ <S>$$
$$2.\ <S> ::=$$

Figure 3

As a result, the case A has an immediate application: the transition to concrete syntax. This takes place if work is done exclusively on the abstract syntax.

The formal system describing the semantical evaluation uses exclusively the string concatenation as evaluation operation.

5. Case B. We shall use the syntax from figure 1a) in order to exemplify the multiple evaluations of nonterminals, hence the case of carriers of the cartezian product type.

Example 2

a)	b)
1. $<S>::=<S><S>$	1. $<S>::=(<S1'>)<S1''>/1<S2'><S2''>$
2. $<S>::=$	2. $<S>::=/2$

Figure 4

Example 2 presents a formal system for describing the evaluation of the nonterminal S into two semantic domains $\{(\,,)\}^*$ and $\{1, 2\}^*$. The evaluations are pairs of symbol strings. We shall separate them with the aid of the metasymbol /. A description in this formal system is:

$$<S1>/<S2> \Rightarrow (<S1'>)<S1''>/<S2'><S2''> \Rightarrow ()<S1>/12<S2>$$
$$\Rightarrow ()\,(<S1'>)\,<S1''>/121<S2'><S2''>$$
$$\Rightarrow ()\,()<S1>/1212<S2>$$
$$\Rightarrow ()\,()/12122$$

The first string represents a sentence in the concrete syntax shown in figure 3, and the second string is first's "left" analysis (i.e. the string of productions whose application produces left derivation of the sentence under consideration). The language used in Figure 4b) uses digits 1 and 2 for specification of both evaluation sets of the nonterminal S : 1 for the set $\{(,)\}^*$ and 2 for set $\{1, 2\}^*$.

Example 3

```
1. <S> : : = <S> <S>        1. <S> : : =(<S1'>)<S1''>/2 + <S2'> +
                               + <S2''>)>
2. <S> : : =                2. <S> : : = / 0
```

Figure 5

Example 3 uses as semantics fields $\{(, (\}^*$ and N (the set of natural numbers). The evaluations consist of the string generated in the concrete syntax of Figure 3 and its length. The evaluating functions are concatenation, for the first evaluation, and addition for the second evaluation.

6. Case C. *Example 4.*

```
1. <S> : : = <S> <S>   1. <S> : =[WL][MR]<S>[ML][WR]<S>
2. <S> : : =           2. <S> : =
          a)                        b)
```

```
char Screen [L] [C] ;
struct { int l, c} Pos ;
int Margin ;
```

c)

```
WL ( )                              MR( )
{                                   {
  if ((Pos.l <L) && (Pos.c<C))      Pos. c +=2 ;
      Screen[Pos.l++][Pos.c]='(' ;      }
}
WR ( )                              ML( )
{                                   {
  if ((Pos.l<L) && (Pos.c<C))         Pos.c− =2 ;
      Screen [Pos.l++][Pos.c]=')' ;  }
}
```

d)

Figure 6

In Figure 6b) is presented a context processing as a semantical evaluation for the syntactical components of the grammar shown in figure a). The context is a data structure described in C language in Figure 6c). It comprises : an array Screen having $L \times C$ characters, a cursor *Pos* characterized by 2 integers and a left alignment limit *Margin*. The context may represent a screen in which we write. The inscribed text contains only the characters (,) and blanks, and represents the indentation structure of a correctly balanced string of paranthesis. For the string (( )) ( ) the screen will appear as in Figure 7 :


```
+----------------+
|                |
|  (             |
|    (           |
|    )           |
|  )             |
|  (             |
|  )             |
+----------------+
```

Figure 7

In the evaluation description language used in Figure 6b), the metasymbols [,] were used to enclose the action primitive names. Primitive description in C language is in Figure 6d).

The derivation of the printing "program" for the screen in Figure 7 is as follows :

```
<S> ⇒ [WL] [MR]<S> [ML] [WR] <S>
    ⇒ [WL] [MR] [WL] [MR]<S> [ML] [WR] [ML] [WR] <S>
    ⇒ ...⇒[WL] [MR] [WL] [MR] [ML] [WR] [ML] [WR] [WL]
      [MR] [ML] [WR].
```

In our example a certain previous context initialization has been assumed. This initialization may be explicitly introduced in the description (Figure 8)

```
0. <Z> : : =<S>           0. <Z> : : = [IN]<S>
1. <S> : : =<S><S>        1. <S>: : =[WL][MR]<S>[ML][WR]<S>
2. <S> : : =              2. <S> : : =
        a)                          b)
```

```
IN ( )
 { int i, j ;
        for (i=0 ; i<L ; i++)
           for (j=0 ; j<C ; j++)
         Screen [i] [j]=' ' ;
        Pos.l = Pos.c = Margin = 0 ;
 }
```

c)

Figure 8

7. The semantical machine. There are some relationships between our formal systems and other known ones. Thus, case A can be integrated into the string translation grammar class [6] and case B into the syntax directed translation schemata class [7]. The formal system introduced in case C represents a particular case of *ATN* [8].

The nonformal manner of introducing formal systems in the four examples presented, as well as the languages used in their description, suggest the existence of a basic *CFG* upon which are grafted utilities corresponding to the three cases. In regard to implementation, it becomes necessary to resort to a formal model different from *CFG*, but equivalent to it in respect to the specified language: the "push — down" automata (*PDA*). One can presume that in each of the above mentioned examples it exists a basic *PDA*, to which we would have to add facilities in order to model as exactly as possible the desired processings. The usual syntax directed processings consist of combinations of the three cases. So, it would be interesting to create an integrating model for the three different types of processings. The basic model automaton will have the role of directing the model evolution. It will command the semantical operators execution in moments precisely determined by the syntactical components. The *PDA* and the model can be compared in a simulative approach: by selecting from the model evolutions only the applications of operations, any evolution of the basic PDA must have at least one correspondent in the model evolutions and, vice-versa, in each model evolution we must find a corresponding evolution of the basic *PDA*.

The proposed model structure is presented in Figure 9. The model is composed of two subsystems: the central unit and the memory.

The central unit has the structure of a *PDA*. The functioning of a *PDA* is usually directed through tables. Here, the tables are composed of *patterns*. A pattern is a symbol string. It results from a production right part enriched with symbols having semantical use. Each pattern is associated to the production nonterminal.

The stack contains pattern symbols. They determine the future evolution of the processing: the symbol at the stack top will determine the subsequent action to be undertaken by the processor. If the top symbol is a nonterminal, it causes a syntactical guidance action: the symbol will be replaced with a selected pattern. The pattern is one associated with the replaced nonterminal and selected on the input basis. If the top symbol is a semantical symbol, a semantical action occurs which will modify one of the memories.

In order to eliminate the nondetermination (useless in the majority of problems and, moreover, difficult to express and achieve in the case of an implementation in a nonconcurrent programming language) the selection is obtained through inputs, which are production numbers of the basic grammar productions. For a correct automata evolution the input production number has to be exactly associated to the nonterminal present at the top of the stack. Thus, the pattern corresponding to the input production will be the one selected.

One observes that the placement of semantical symbols in the stack can be avoided because they do not imply recursion as nonterminals.

As is known, the stack is necessary because it allows the maintaining of the evolution "track", represented by the string of productions used in the nonterminal expansion and partially solved. Each such production is marked with the place where it was suspended.

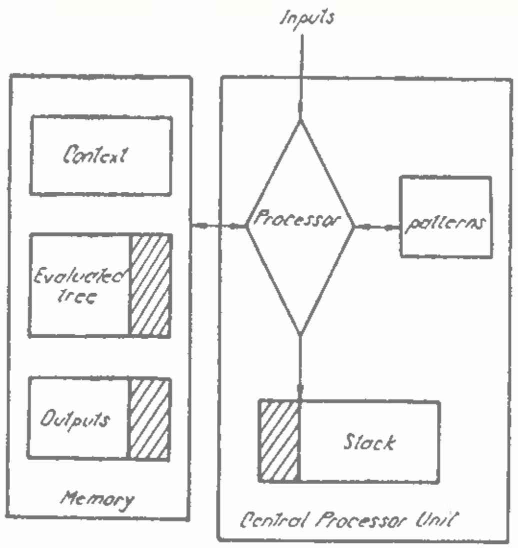

Figure 9

This is why symbols from the stack can be replaced with pairs "(production, place in production)" (as in the case of the analysis of *LR* — grammar based languages). Of course, the processor that manages such a stack will have at its disposal the full pattern collection, including the possibility of referring any pattern symbol. We shall return to this model variant when we present its implementation [9].

In accordance with the access mode, the memory is structured into: outputs, evaluated tree and context memories.

The outputs memory contains output symbol sequences placed by the processor; the respective memory can be implemented as a file.

The evaluated tree memory is an arborescent structure isomorphic with the forest of completely evaluated syntax trees, those who have directed, up to now, the processing (in other words, the syntactical components eliminated from the *PDA* stack). In the structure nodes, in the place of the nonterminals from syntax trees, there are tuples belonging to the sorts associated with the respective nonterminals, by way of the operation of embedding of the syntax in the semantic field.

The context has a usually nonhomogenous structure, with direct access to the components.

The actions upon the memory are launched when the top of the stack contains a symbol with semantical use. By specializing the symbols according to the type of memory, affected by action, we shall obtain a better systematization, but we will eliminate what in practice is acceptable and frequently encountered: the possibility of a combined action upon two or three types of memory.

The moments for action lauching are dictated by the positions occupied in patterns by the symbols with semantical use. Therefore these moments depend upon the mode in which the right parts of the productions are "scattered" with symbols of semantical use.

Extremely important in pattern design is the choice of the place that the symbols that start the evaluation of nonterminals by sorts, occuppy. A brief discussion is of order.

The proposed model simulates *PDA*-s having advance and expansion actions, in other words with a top-down strategy for passing the syntax tree [9]. Thus, if we also add the condition that the tree be evaluated in a single passing, naturally in an ascending manner, the need for L-attributation results, thus the evaluation of any nonterminal sort from the left side of the production only after all the nonterminals from the right side of the production whose values are dependent on it have been evaluated. It follows that the evaluation function symbols of the different nonterminal sorts from the left side of the production, must be placed, as part of the pattern, at the right of the nonterminals that take part, by way of different evaluations to the computation described by the function. As a rule, a solution that is allways valid, is that of placing these symbols at the end of the pattern. On launching the actions that accomplish the evaluation functions, in the evaluation tree will certainly exist all the trees corresponding to the nonterminals from the right of the production whose processing is ending. As a result of the evaluation, a new tree in the tree memory is created with the root corresponding to the evaluated nonterminal and as descendents the tree with the roots corresponding to the right part nonterminals. It is observed that these roots will be definitely masked as of now by the new one. That is why, in a more efficient implementation and one in which evaluated trees are no longer necessary, but only the starting nonterminals values, the nodes that are not further tree roots can be even eliminated from the memory.

8. The formal model. Without insisting too much upon the formalization of the proposed model, we shall introduce its main elements.

— the state set, $Q=\{q_1, q_2, q_3\}$ where q_1 is the working state,
q_2 is the error state,
q_3 is the acceptance state

— the input set, $\Sigma=\{1, 2, ..., n\}$, where n is the number of productions

— the nonterminals set, N; there is a function:
assoc : $\Sigma \to N$, that associates a nonterminal to each production,

— the output symbol set, Δ,

— the nonterminal evaluation symbol set, E,

— the context primitives symbol set, C,

— the transition functions, δ,

$$\delta : Q \times (\Sigma \cup \{\lambda\} \times \Gamma \to Q \times \Gamma^* \times \Delta \times F \times G,$$

where

$\Gamma = N \cup \Delta \cup E \cup C$, is the pattern symbol set,

$F = \{(M(A_1) \times ... \times M(A_k) \to M(A);$

$(A_1, ..., A_k; A) \in N^* \times N, 0 <= k <= \text{Card } (N)\}$

the nonterminal evaluation function set

$G =$ (Cont$\to$Cont), the context primitive set.

The following notations have been used:

$(X \to Y)$, the set of partially defined functions over X, with values in Y,

$M(A_j)$, the carrier of the nonterminal A_j,
Cont, the set of context values,
λ, the empty string.

The definition of δ should point out the different working conditions: the expansion by way of patterns, output generations, nonterminal evaluations and context processings. For example, expansion by way of the pattern $\alpha \in N^*$, corresponding to production i, associated with nonterminal A, can be definied as:

$$\delta(q_1, i, A) = (q_1, \alpha, \lambda, O_F, I_G),$$

where O_F is the null function from F,
I_G is the identical function over the context.

One could also define: the configurations and the relations beetween configurations, the initial and final configurations and the model's evolution.

9. The model implementation. In order to implement the proposed model, as a suport for syntax directed processings we choose a *C* language solution. The model implementation is presented in Appendix. The implementation followed in 2 steps:

i) First, we established the necessary data structures for the different components implementation. In Appendix we used the following implementation variant for these structures:

The Pattern Memory. Array of records. It shall be initialized with patterns grouped by associated nonterminals.

The Stack. Array of pairs (number of production, position) or, as in the variant presented in [7], only an array of positions in the pattern memory.

Outputs. Text type file.

Evaluation Tree. The list of trees in whose nodes are present records, containing nonterminal values and information regarding linkage between nodes etc.

The Context. A certain given structure (probably dynamic), specially designed for each application.

Certain structures, such as the evaluation tree and the context memories and, eventually, the processor stack, require working primitives depending upon the chosen data structures. This is the case of the functions present in *F* and *G*. For the rest of the structures, the facilities made available by the language through indexation, selection, derefferencing etc. are sufficient.

ii) Secondly, the processor was implemented. In principal, this meant the implementation of the transition function. Schematically, the structure of the processor is a cycle exited when the state becomes either q_2 or q_3. Inside the cycle, according to the symbol present at the top of the stack, the different cases of its definition will be selected and a model imposed processing will be achieved. For example, in the case of a nonterminal present at the top of the stack, the input will be taken over (from a file buffer or from the console), the input will be verified whether it has associated the respective nonterminal, the pattern will be indentified according to these informations, and afterwards the pattern is copied into the stack.

An efficient implementation variant presumes the existence, in the stack of only the positions in the temporarily abandoned patterns and thus direct work upon the pattern memory. In this case, the processor memorizes the current position in the pattern memory and the selection decision is taken according to the symbol found at that position. If it is a nonterminal, the subsequent position in the pattern memory is memorized in the stack (for return, at the reusal of the pattern), and the current position is moved to the first pattern symbol indicated by the input (if its association with the nonterminal verifies).

10. Final considerations. The paper presents a possible approach to syntax directed processing. Accepting as a starting premise the fact that such processings are evaluations in a wide variety of semantics fields of *CFL* syntactical structures, an automata for such language oriented processing has been introduced. This model performs three distinct categories of syntax directed processings, most commonly encountered in practice.

The model is centered on a *PDA* that works with tables. Its syntactical actions are extended by semantical ones that work with more or less particularly structured memories containting values from different semantics fields, having more or less particular structures.

The model, which has been extensively presented from the physical point of view and briefly from a formal point of view can be easily implemented starting from its structure and functioning. The complete model implementation is also presented.

The proposed model crystallized during authors' work occasioned by the elaboration of a program development system [10]. Based on the model some processors were implemented: syntactical editors and analysers, pretty-printers, converters from one representation form to another. From the beginning, the possibility become apparent for grouping the processings, directed by syntactical structures belonging to the same language, irrespective of aim of value domain of the processing. Afterwards, it was observed that all the descriptions of these processings could be "mixed" into a unique representation based on patterns, having as their common points these syntactical structures. By changing only the semantical evaluation procedures one could comute from one type of processing to another. The program structure was composed of 1) a kernel, whose main purposes was that of pattern crossing and maintaining the "trace" of the program execution, and 2) program segments containing semantical procedures that were loaded when the type of semantical processing accomplished by these procedures was to be started.

The restriction to have as inputs only production numbers, correctly ordered in a left analysis of a language sentence can be partially eliminated by introducing a suplimentary layer of the model, which is meant to generate correct input strings, starting from some more general commands, expressed in a more powerful language. Also, such a layer can accept queries about what productions could be used, according to a given nonterminal present at the stack top. Once such a production is chosen, it can be transmitted to the model input. Or, there can be commands that generate a sequence of inputs until a certain condition of a syntactical nature is achieved (for example: the occurence at the top of the stack of a certain nonterminal).

Finally, the present approach facilitates the elaboration of generators for language oriented processing programs. The one elaborated by the authors of this paper accepts as input the abstract syntax enriched with semantical symbols. The generator carries out the pattern collection, a standard procedure for the processor simulation and a collection of files containing sort descriptions, context structures, action primitives, evaluation functions, all of them organized as a program. Most of this information is obtained interactively in a user-friendly oriented manner.

The authors stake no claim of having exhausted the varied multitude of aspects and facilities brought about by the syntax directed processings. They hope to attract attention, once again, to this important class of processing.

Appendix

```
/*****************************************************************
*                                                                *
*                    P R O C E S S O R . C                       *
*                                                                *
*****************************************************************

#include <stdio.h>
#define EOL 0
#define NET 1
#define TRM 2
#define ACT 3
#define OUT 4
#define _M_ 100
char * _E_[ ] = {
            "Bad terminal",
            "Stack overflow",
            "Bad production number"
            };
struct Patt_mem {int op, arg; };         /* pattern memory type */
struct Net_limit {int low, high; };
extern struct Patt-mem Memory [ ];            /* pattern memory */
extern struct Net_limit Net_lim [ ];     /* neterminals→
                                              productions map */
extern char * (*Act [ ]) ( );                /* action symbols */
extern char * (*Atr [ ]) ( );        /* attribute type declarations*/
extern char * Out_tab [ ];           /* output symbols          */
extern int   Prod_map [ ];
int process ( )
/* returns 1 for success or 0 for failure */
{int ip,          /* instruction pointer */
     si,          /* stack index         */
     s[_M_],            /* stack         */
```

```
q, a,  /* actual opcode and argument*/
e,               /* error code        */
x ;                /* input code      */
ip = 0 ;            /* xfr address */
si=—1 ;  /* stack is empty initially */
for (; ;)  {
q = Memory [ip] . op ;
a = Memory [ip++] . op ;
switch (q)  {
  case TRM :  if (usrinp ( )  != a) { e = 0 ; goto _err_ ; } break ;
  case NET :  if (++si >=_M_) { e= 1 ; goto _err_ ; }
              s [si] = ip ;
              x = usrinp ( ) ;
              if (x < Net _lim [a] . low || Net_lim [a] . high > x)
                  { e = 2 ; goto _err_ ; }
              ip = Prod_map [×] break ;
  case OUT  : usrout (Out_tab [a]);break ;
  case ACT  : (*Act [a]) ( ) ; break ;
   case EOL : (*Atr  [a])  ( ) ; if (si < 0) return (1) ;
              ip = s [si- -] ; break ;
  }
 }
  _err_ :   ;
            fprintf (stderr, "*PROCESSOR*\007 %s\n", _E_ [e]) ;
            return (0) ;
}
```

REFERENCES

1. Irons E. T. — *A Syntax Directed Compiler for ALGOL-60*, Comm. of ACM nr. 4, 1961
2. ADJ, *Initial Algebra Semantics and Continuous Algebras*, J. ACM, v. 24, nr. 1, 1977
3. Knuth D. — *Semantics of Context-free Languages*, Math. Systems Theory, 1968
4. C.A.R. Hoare — *Consistent and Complementary Formal Theories of the Semantics of Programming Languages*, Acta Informatica, 3, 1974
5. McCarthy J. — *Towards a Mathematical Science of Computation*, Proc. IFIP Congress, North-Holland, 1962
6. Lewis P.M., Rosenkrantz D. J., Stearns R. E. — *Compiler Design Theory*, Addison-Wesley, 1976.
7. Aho A.V. Ullman J. D. —*The Theory of Parsing*, Translation and Compiling, vol. II, Prentice Hall, 1973
8. Woods D., — *Augmented Transition Networks*, CACM, 1970
9. Şerbanaţi L.D.—*Limbaje de programare şi compilatoare*, Ed. Academiei, R.S.R. ,1987
10. Colectiv mixt IPB — ICSIT-TCI — *Elaborarea unui sistem de dezvoltare asistată de calculator a programelor*, Contract de cercetare ştiinţifică, Documentaţie de prezentare, 1987

Received 1.II.88

Polytechnical Institute of Bucharest
and
Research Institute for Computer Tehnology and Informatics
Bucharest

RECENZII

I. M. GELFAND: *Collected Papers*, Volume II, Editors: Gindikin, S. G., Guillemin, V. W.; Kirillov, A. A.; Kostant, B.; Sternberg, S., Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong-Kong, 1988, 1039 p., D M 248.

Al doilea volum (din cele trei, cuprinzînd studiile complete, ale unuia din cei mai remarcabili matematicieni ai secolului XX, I. M. Gelfand), conţine studiile şi articolele referitoare la teoria reprezentărilor unor clase foarte generale de structuri algebrice. Această problematică a constituit una din cele mai importante din toată activitatea de peste 40 de ani de cercetare a renumitului savant.

Rezultatele obţinute în această direcţie sînt judicios compartimentate în nouă secţiuni, fiecare dintre acestea cuprinzînd direcţii, esenţial noi, în strînsă legătură cu alte ramuri ale matematicii sau fizicii. Cele mai multe din lucrările prezentate sînt rodul colaborării cu reputaţii matematicieni D. A. Rajkov, M. A. Naimark, M. I. Graev, D. A. Kazhdan, I. N. Bernstein şi A. V. Zelevinskij, care s-au format la şcoala din cadrul seminarului inaugurat şi condus peste 40 de ani de prestigiosul matematician Gelfand.

Contribuţii fundamentale în acest domeniu au adus matematicieni de reputaţie mondială, dintre care remarcăm pe E. Cartan, H. Weyl, Harish-Chandra. A. Selberg, A. Weil, R. Langlands şi P. Deligne.

Dar, contribuţiile lui I. M. Gelfand, atît ca profunzime în abordarea problemelor, cît şi ca frumuseţe a rezultatelor, sînt cu mult superioare.

Cele nouă părţi ale volumului: Probleme generale ale teoriei reprezentării; Reprezentări infinit dimensionale ale grupurilor Lie semisimple; Geometria spaţiilor omogene; Modele de reprezentări: reprezentări de grupuri peste diverse cîmpuri; Verma-module: rezoluţii ale reprezentărilor finit-dimensionale; Algebre înfăşurătoare şi corpurile lor de fracţii; Reprezentări finit-dimensionale, Reprezentări nedecompozabile ale grupurilor Lie semisimple şi ale algebrelor finit-dimensionale: probleme de algebră liniară şi Reprezentări ale grupurilor infinit-dimensionale, pun în evidenţă, cu pregnanţă, atît aria vastă a problemelor abordate şi a tehnicilor şi metodelor utilizate cît şi profunzimea rezultatelor obţinute.

Dintre ideile originale (de mare generalitate şi fineţe) referitoare la această problematică, se remarcă: reprezentările unitare infinit-dimensionale ale grupurilor clasice admit descriere asemenea reprezentărilor finit-dimensionale; fundamentul cel mai natural al analizei armonice pe un grup semisimplu este un inel comutativ şi importanţa geometriei integrale în teoria reprezentării.

Remarcăm că „baza Gelfand-Tsetlin" pentru reprezentările finit-dimensionale ale grupurilor clasice a găsit un deosebit ecou în lumea matematicienilor, atît prin importanţa ei teoretică, cît şi prin aplicabilitatea largă în fizica teoretică.

Volumul, care se încheie cu Apendix şi Conţinut al volumelor I—III, se adresează matematicienilor şi cercetătorilor din domeniile fizicii teoretice, oferindu-le un bogat şi fundamental material instructiv.

N. Luca