LIBERTAS MATHEMATICA , VOL. VII (1987)

TICS System: A Translator Generator

Teodor Rus

Abstract:

The early compiler-writing systems provide a rich programming environment that is well tailored to writing a compiler by hand. Since the translation (compiling) activity in a computer installation represents an essential part of the software system, a true technology for the translator implementation is still sought. This technology should enclose the automatic compiler generation as well as the automatic translator generation. The automatic translator generation is required as a solution for the reusability of the huge experience of program writing using various programming languages which might not be available on a certain computer installation.

The general approach developed so far for an automatic compiler generation [3, 9, 19, 37] are usually applied for implementing a compiler that maps a a source programming language PL into a target machine language ML and fits in the following framework:

1. A formal definition of the syntax of the PL is chosen and a semantics (formally or informally specified) is then associated with it.
2. The language definition at 1 above is transformed into an implementation oriented definition. I.e., this definition is transformed into a computing structure and an abstract machine which is capable to perform the computations provided in that computing structure. The common computing structure is an abstract syntax tree and a machine that can compute it [2, 36]. This transformation is called the *front end* of the compiler.
3. The computing structure developed at 2 needs to be automatically transformed into a program for an actual machine. The algorithm performing this transformation is the *back end*, i.e. the code generator of the compiler.

This approach does not accommodate the more general problem concerning the automatic translator generation, in which target language is not necessarily the machine language of a certain machine. To address this issue the TICS system is based on a somewhat different approach. This approach consists of providing a formal tool for specifying both programming language and compiler as mathematical objects in the framework of universal algebras. A universal construction in algebra is then applied to specify the compiler construction as an algorithm over the class of languages defined by a given pairs of specifiers. The paper discusses the theoretical basis of this construction as well as some implementation results.

Following TICS approach the Tower of Babel of Languages available on a given computer installation can be replaced by a mechanism which offers to the user a chance to specify and implement his own programming language, according to his own computing needs.

1. Introduction

Compiler construction represents one of the most attractive topics in computer science because of the great need for compilers and the formal features of the objects involved in compiler construction. Since a compiler maps a source language into a machine language, in order to define a compiler a clear concept of programming language is required. We use the following algebraic concept of a programming language:

Definition_1: A programming language is a triple

$$PL = \langle Sem, Syn, learn : Sem \longrightarrow Syn \rangle$$

where *Sem* is the language semantics, *Syn* is the language syntax, and *learn* is a learning process that allows the user to handle abstract and/or concrete computing objects in *Sem* using their symbolic expressions in *Syn*.

Since a programming language is used as a communication mechanism, symbols in *Syn* are handled between communicators using such a language instead of the computing objects in *Sem* these symbols stand for. Therefore, for communication purpose the assumption is that there exists another function denoted by *eval* that is used to evaluate symbols in *Syn* to their value in *Sem*, i.e., $eval : Syn \rightarrow Sem$ such that $learn \circ eval = 1_{Sem}$. The concept of a compiler specifies a mapping from a high level programming language into a machine language. The concept of a translator is more general and specifies a mapping from a programming language into another programming language and is defined as follows:

Definition_2: Let PL_1 and PL_2 be two programming languages,

$$PL_1 = \langle Sem_1, Syn_1, learn_1 : Sem_1 \rightarrow Syn_1 \rangle$$

$$PL_2 = \langle Sem_2, Syn_2, learn_2 : Sem_2 \rightarrow Syn_2 \rangle$$

A translator $T : PL_1 \rightarrow PL_2$ is a pair of functions $T = \langle H, C \rangle$,

$$H : Sem_1 \rightarrow Sem_2$$

$$C : Syn_1 \rightarrow Syn_2$$

that make the following communication diagram commutative:

$$\begin{array}{ccccc} & learn_1 & & eval_1 & \\ Sem_1 & \longrightarrow & Syn_1 & \longrightarrow & Sem_1 \\ \downarrow H & & \downarrow C & & \downarrow H \\ & learn_2 & & eval_2 & \\ Sem_2 & \longrightarrow & Syn_2 & \longrightarrow & Sem_2 \end{array}$$

While the H component of the translator is an abstract map, the C component is an actual map that transforms well formed expressions in Syn_1 into appropriate well formed expressions in Syn_2 such that their meaning is preserved. The algorithm that performs this mapping is composed of two parts: a recognizer, which recognizes the objects in Syn_1 as valids well formed symbolic constructs, and a generator, which regenerates the objects validated by the recognizer as well formed symbolic constructs in Syn_2.

2. Conventional Compiler Construction

The implementation of compilers for conventional programming languages use a recognizer that generates an intermediate representation of the symbolic constructs recognized by it which is suitable to be used by the generator. The design of a language recognizer that employs a grammar as mechanism for source language specification uses a tree language as the intermediate representation. This is appropriate because the process of a program derivation from the grammar can be expressed by a syntax tree whose nodes can be decorated with the values of the constructs represented by the subtrees rooted in that nodes. On the other hand, algorithms [2, 36, 37] are available that can map a tree representing a computing process into a machine language program that performs the computation attached to its root. Thus, the following diagram represents a compiler generated in this framework:

$$Syn_1 \xrightarrow{Recognizer} Abstract\ Syntax\ Tree \xrightarrow{Generator} Syn_2$$

where Syn_2 is necessarily the language of a given machine.

The main history of compiler construction can be summarized by the following three understanding stages of the language and compiler as formal objects handled in the above diagram:

1. **Stage_1:** Simply write compilers to "translate" programs. Both recognizer and generator are ad hoc constructs at this stage.
2. **Stage_2:** Develop special languages for writing compilers. The activity of writing a compiler at this stage is directed by the types of data and operations experimentally identified in the work of writing compilers at **Stage_1**. The components of the compiler, recognizer and generator, are still ad hoc constructs. However, an appropriate language to express operations and data involved in these constructs is provided.
3. **Stage_3:** Syntax directed compiler construction [3]. The activity of writing a compiler at this stage is based on a formal device for specifying the source language syntax. Therefore, the recognizer is mathematically specified by an appropriate automaton provided by the syntax specifier. The generator is however identified with the algorithm that decorates the syntax tree constructed by the recognizer with the appropriate semantics attached to the syntax rules.
4. **Stage_4:** Semantic directed compiler construction [16]. The activity of writing a compiler at this stage is based on a formal device for specifying both syntax and semantics of the source language. The recognizer is as specified at the **Stage_3**. However, the generator is an automaton that is capable to synthesize the semantics associated with the nodes of the syntax tree constructed by the recognizer as an expression in the target language [9, 16, 20, 21, 23, 24, 25]. Since the target language is not actually accommodated by this scheme, confusion concerning the the mapping of the computations synthesized by the generator into the target machine is created at this point. However, since algorithms for tree computation with actual machines are

available [2, 36, 37] these algorithms can make justice to semantics directed compiler construction. But it turns out that **Stage_3** and **Stage_4** are actually the same.

No one of the stages identified above in the history of compiler construction is based on the explicit formal specification of both source and target languages involved in the compiling relation and their association by a translation relation mathematically defined. However, tremendous amount of work for the foundation in this field has been done and many results are reported in a great number in various journals, books, reports, letters, etc [3, 6, 7, 16, 19, 37]. Their number is still increasing and sometimes it is easier to rediscover a particular feature used in compiler construction instead of using it as it likely has been developed and already reported somewhere. Therefore many approaches, algorithms, applications, etc., are anonymous, constituting a kind of folklore in computer science and it is hard to refer them by the author's name. In order to contrast our approach for compiler construction with the conventional approaches presented above we discuss further in somewhat more details only the semantic directed compiler construction.

The general mechanism on which a semantic directed compiler generator is based consists of a concept of a programming language specified by three distinct mechanisms:

1. BNF notations for syntax syntax specification.
2. Attribution (functional and/or denotational) for semantics specification.
3. Syntax and Semantics association into a language by arbitrarily associating attributes in semantics with syntactic constructs in the syntax.

Thus, the compiler constructor sees a programming language as a triple

$$PL = \langle Syntax, Semantics, eval : Syntax \rightarrow Semantics \rangle$$

where *Syntax* is given in advance by means of a finite set of BNF rules which define the collection of well formed symbolic constructs allowed in the language. *Semantics* is associated with the *Syntax* as computing abstractions specified by semantics domains and semantics functions through the mechanism of attribution rules and/or denotations [6, 13, 40].

The formal mechanism that specifies the syntax of the programming language is a context free grammar defined by a finite set of BNF rules. This grammar is defined as follows:

Definition_3: Let $R = \{r_1, r_2, ..., r_p\}$ be a given finite set of BNF rules. Each $r \in R$ has the form $r = A \longrightarrow t_0 A_1 t_1 ... t_{n-1} A_n t_n$ for $n \geq 0$, where capital letters denote parameters and lower case letters denote fixed strings. The context-free grammar specified by R if the quadruple $\mathbf{G(R)} = \langle V_n, V_t, R, S \rangle$ where:

1. V_n is the finite set of symbols used as parameters in the rules in R and are called nonterminals.
2. V_t is the finite set of fixed strings (key words) used in the rules in R and are called terminals. $V_n \cap V_t = \emptyset$. The set $V = V_n \cup V_t$ is called the language vocabulary.

3. R is the finite set of BNF rules considered as the rewriting rules (or derivation rules) in G(R). The rewriting (or derivation) relation is defined as follows: if $x \in V^*$, and $x = x_1 A x_2$ for $x_1, x_2 \in V^*$, $A \in V_n$ then if there exists $r \in R$ and $r = A \longrightarrow t_0 A_1 t_1 ... t_{n-1} A_n t_n$ where $A \in V_n, A_1, ... A_n \in V_n \cup \{\epsilon\}, t_0, t_1, ..., t_{n-1}, t_n \in V_t \cup \{\epsilon\}$ then x can be rewritten as $x \Rightarrow x_1 t_0 A_1 t_1 ... t_{n-1} A_n t_n x_2$. The transitive and reflexive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$ and is called the derivation (rewriting) relation of the grammar.

4. $S \in V_n$ and it represents the start symbol or the axiom of the language.

The language L(G(R)) generated by G(R) is identified with the syntax of the programming language and is defined by the equality:

$$L(G(R)) = \{x \in V_t^* | S \Rightarrow^* x\}$$

In order to define the *Semantics* components of the language, [6, 7, 9, 12, 13, 16], for every symbol $A \in V_n \cup V_t$ a set of inherited attributes I(A) and a set of synthesised attributes S(A) are given and $I(A) \cap S(A) = \emptyset$. For each $d \in I(A) \cup S(A)$ an abstract object called the *semantics domain* associated to d and A and denoted by D(d, A) is considered. For each derivation rule $r = A \longrightarrow t_0 A_1 t_1 ... t_{n-1} A_n t_n \in R$ a set of semantics rules given in the form of functions

$$D(d, A1) \times D(d, A2) \times ... \times D(d, An) \rightarrow D(d, A)$$

which define the values of an attribute $d \in I(A) \cup S(A)$ in terms of the values of other attributes are provided by denotational semantics.

The relation between the symbolic forms in L(G(R)), their derivations by means of G(R) and the associated semantical objects is sketched in figure 1.

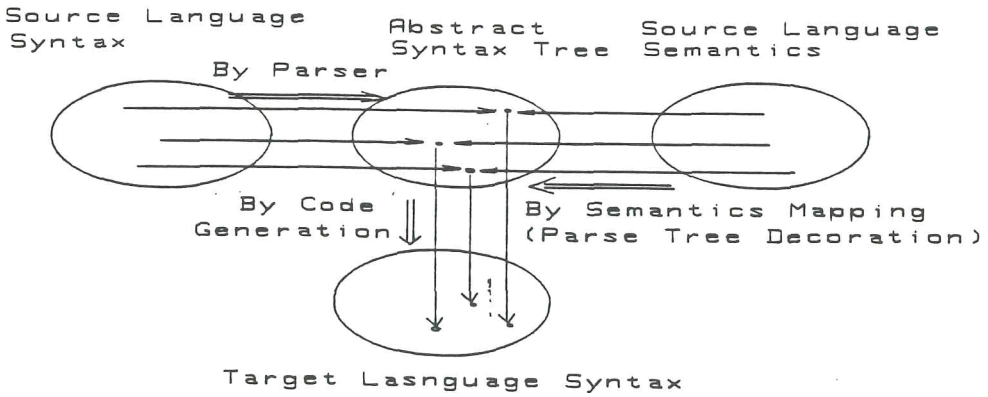

Fig. 1

Almost all compiler generators designed in this framework have as input data a grammar given as stated above and as output a compiler which acts as follows:

Step_1: Each program $P \in L(\mathbf{G}(R))$ is regenerated as a derivation tree using the derivation process defining the language $\mathrm{L}(\mathbf{G}(\mathrm{R}))$. This is achieved by a parser which can be mechanically produced having an algorithmical nature.

Step_2: The derivation tree obtained at **Step_1** is decorated by attaching to its nodes the values provided by the semantic rules associated with the derivation rules that generated them. The decorated tree thus obtained represents a computing process specified in terms of the denotational functions attached to its nodes and acting in the computing environments specified by the semantic domains. The meaning of the program is represented by the values of the denotation functions attached to the root.

Step_3: The decorated tree obtained in **Step_2** is then regenerated as a new syntactic construction in terms of a new programming language called target language.

In the **Step_3** above the language provided by the instruction set of a given machine is most often considered. Since algorithms that can map expression trees into machine computations are known [2, 36, 37] these algorithms are used to regenerate the tree in the language of a given machine. The process of the tree regeneration used at this step is called the code generation mechanism for the chosen machine.

Before discussing the limitations of this approach we illustrate it with a sketch of the compiler generator reported in [23] which acts as shown in figure 2.

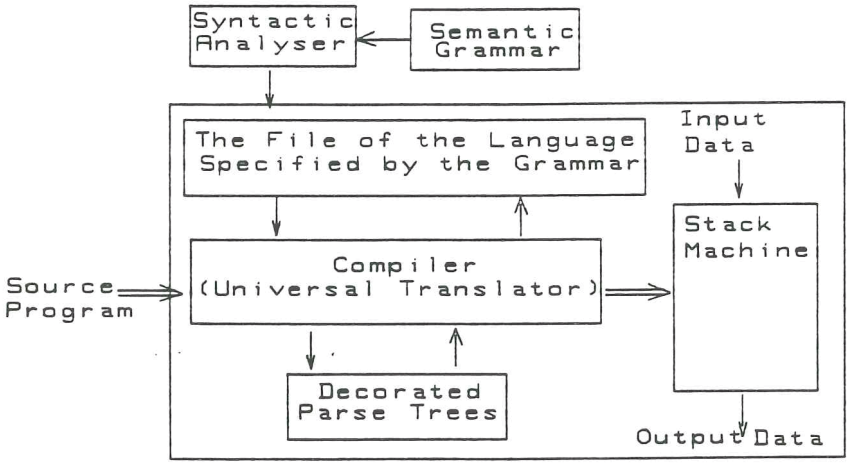

Fig. 2

The rectangle encloses the compiler generated by this approach.

3. Limitations

The main limitations of the semantic directed compiler generation approach discussed above are:

1. The approach is not based on a concept of a compiler that is mathematically defined. The specification mechanism on which this approach is based concerns only the source language. Sometimes the target language is implicitly specified by the the semantics functions attached to the rewriting rules. The PERLUETTE generator discussed in [10, 11] explicitly considers the target language but only as an unspecified object. Therefore this limitation results in obscuring **Step_3** above.

2. **Step_3** is basically directed by the source program P. Since P is only an expression form of a given computing process it follows that the given computing process is not freed of its source form of representation, namely P. Moreover, P is regenerated by the compiler using the derivation process defined by a grammar while the programmer generates P in an algebraic manner. Therefore, the actions associated with **Step_3** are dependent on the grammar specifying the source language and thus they cannot be seen as an universal code generator depending only upon the target language specifier exactly like the parser is an universal algorithm depending only upon the source language specifier.

3. Only the syntax component of the language is defined as a formal object. Even when the semantics component of the language is formally specified by mathematical functions (denotations) [13, 16], or by algebraic constructions [1, 5, 6, 12, 35], it is artificially associated with the syntax. It explains more or less formally the computing process behind the syntactic derivation rules. On the other hand, when a programmer writes a program, it expresses a given computation object using syntactic rules of well formation. He/she does not write the program and then associates it with its meaning.

The denotational semantics [5, 12, 13, 16] concerns the designing of denotations for constructs. It is possible to use the notation of denotational semantics as specifying a translation from one language, the language being described by the syntax, to another language, the metalanguage consisting of the notations described in denotational semantics. But this metalanguage is ad- hoc and not really proper defined. It is possible further to formalize it [5, 12, 20, 21]. But then the new problem is that we translate from a high level language (the syntax), well understood by programmers, into another high level language, (the language of algebra), less understood if not understood by the programmers. However, a translator from the language of algebra into a machine language poses the same problems as the original translator posed. On the other hand, trying to discover the algorithm of compilation, the syntax itself can be seen as an algebraic structure generated by a grammar [14, 30]. I.e., the grammar specifying the syntax can be taken as a signature (or a basis) for a class of algebras [33]. The result is that we can generate the language algebra from the syntax rather than generating it from the denotations specifying the semantics. Hence, it follows that by this approach we in fact take as semantics just the formal syntax of the language.

On the one hand by formalization the denotational semantics can be transformed into an algebra of symbols called the algebra of denotations. On the other hand in the class of algebras of a given type there exists an universal object called word algebra [8, 15, 18] which

has the generic property to be a model for all the algebras of the respective class. It is the initial object [1, 8] in the category of algebra specified by that class. The syntax algebra of the language is just the word algebra of the class specified by the grammar when seen as a signature. Thus, it is indeed the model for the denotations. On the other hand the need to construct compilers for programming languages which are defined only by means of their syntax component leaded to the concept of abstract syntax [39]. This is an abstract algebraic structure defined on a collection of symbols. When instead using the grammar as language syntax specifier we use the context-free algebra defined by the respective grammar [14] then the obtained syntax algebra is again the word algebra generated by the algebraic type or basis B(G) (or signature) supplied by the considered grammar G. Hence, it follows that from both sides, starting with a formal semantics as given by the denotational approach or with a formal syntax as given by a grammar approach, by means of structuring using the algebraic machinery we obtain the same universal mathematical object: **the word algebra of the given class of algebras**. From these observations a number of practical suggestions follows:

1. The first suggestion is the programming language definition used in this paper. Namely, for defining a programming language we associate a given computing object called semantics with its syntax representation [4] instead of associating a formal language (the syntax) with an algebraic object called semantics. In this way we provide a natural representation for a given abstract and/or concrete computation increasing readability and understandability of our programming languages. This approach is formally provided by the language of the algebra as an alternative to the grammar for language specification [33]. It is a mechanism which accommodates syntax, semantics, and their association into a language into the same framework. The semantics is an abstract computing object and the syntax is a universal model of that abstract computing object. Thus, it allows us to handle naturally abstractions by means of their symbolic representations, as mathematicians do from immemorial times.

2. The second observation concerns the apparent contradiction obtained by associating a semantics with a given syntax into a language. As we have seen above, formalizing both semantics and syntax we end in the same object: **the word algebra of the given class of algebras**. Since the algebraic structure of the syntax is the word algebra which represent the universal model of the given class, it is quite natural to rediscover it when the syntax and/or the semantics are formalized as algebraic objects. This apparent contradiction is often present in mathematical constructions. It is neglected because in mathematics the association between abstract objects and their names is supposed to be one - one and is obtained by a factoring process. This identification is required by mathematical constructions due to the ideal character of the mathematical objects. In computer science such an idealization could be troublesome. For the definition of a programming language we need as semantics the actual computing object and not its idealization.

4. Theoretical Ideas Behind TICS

TICS is a short form for the expression "Technology for Implementing Computer Systems". It consists of a data base which represents the algebraic specification of a source language and a target language and their association by a translation relation mapping the source language into the target language. A universal algorithm is defined on this data base. This algorithm maps source programs into their target form preserving their semantics. Thus, when this data base is updated with two languages the universal algorithm becomes a translator from the source language into the target language.

To understand TICS's philosophy we discuss further only the main ideas which are behind its construction.

As in the case of a semantics directed compiler construction, TICS is based on a formal concept of programming language. However, TICS considers a language as a communication device in a communication environment. Therefore, the concept of a programming language is seen by TICS at two levels:

1. At the specification level the programming language is seen as an ordered triple

$$PL = \langle Semantics, Syntax, f : Semantics \rightarrow Syntax \rangle$$

Semantics is previously given as an actual abstract or concrete computing system. For example, Peano's arithmetic, an abstract algebra, a heterogeneous abstract algebra, the computing system defined by STRAWMAN for the specification of Ada [38], [1] are such computing objects. *Syntax* is a collection of symbolic constructions which allow us to represent objects in *Semantics* so that to handle concrete forms of representation instead of the abstract computing objects represented by them. f is a learning function or better yet a learning process which associates representation forms in *Syntax* with abstract objects in *Semantics* represented by them. In the case of natural languages f is brain encapsulated.

2. At the communication level, i.e., when the language is used since it is now defined, the language is seen as a triple

$$PL = \langle Semantics, Syntax, e : Syntax \rightarrow Semantics \rangle$$

where e evaluates expressions (i.e., programs) in *Syntax* to their meanings in *Semantics*.

The flow of information during communication using such a language follows the diagram:

$$Semantics \xrightarrow{f} Syntax \xrightarrow{e} Semantics$$

where $f \circ e = 1_{Semantics}$.

Let PL_1 and PL_2 be two languages by means of which a communication is engaged. If $PL_1 = \langle Sem_1, Syn_1, f_1 : Sem_1 \rightarrow Syn_1 \rangle$ and $PL_2 = \langle Sem_2, Syn_2, f_2 : Sem_2 \rightarrow Syn_2 \rangle$

[1] Ada is a trademark of the U. S. Department of Defense

then to achieve this communication the following communication diagram must be commutative:

$$\begin{array}{ccccc} & f_1 & & e_1 & \\ Sem_1 & \longrightarrow & Syn_1 & \longrightarrow & Sem_1 \\ \downarrow H_1 & & \downarrow T_1 \uparrow T_2 & & \uparrow H_2 \\ & f_2 & & e_2 & \\ Sem_2 & \longrightarrow & Syn_2 & \longrightarrow & Sem_2 \end{array}$$

The horizontal arrows are given in the definition of the two languages involved in the communication. The arrows H_1, T_1, T_2, H_2, are not yet defined. However, to perform a communication session these arrows must be constructed such that the commutativity of the communication diagram is ensured. H_1 and H_2 are abstract functions whose existence need to be proved while T_1 and T_2 are concrete transformations of linguistic expressions. They are called translators.

The TICS approach for compiler generation is based on a formal mechanism to specify languages as ordered triples of mathematical objects $Semantics$, $Syntax$, $f : Semantics \rightarrow Syntax$ defined in the same framework. For the TICS system the *Semantics* is an abstract algebra, the *Syntax* is the word algebra generated by the signature of *Semantics*, and f is a homomorphism between *Semantics* and *Syntax*. Thus, the language is seen by TICS as an ordered triple of mathematical objects

$$PL = \langle Semantics, Syntax, f : Semantics \longrightarrow Syntax \rangle$$

as specified by the Definition_1.

The general algebraic mechanism for language specification employed by TICS was called HAS-Hierarchy in [31] and is defined as follows:

1. A base (or signature) B is given. It represents a kind of genetic program of the objects specified by B. These objects are the class of abstract algebras of type B denoted by $C(B)$. Let H(B) be one of these algebras.
2. Let $W(B)$ be the word algebra generated by B in terms of a given collection of symbols. Then for every $H(B) \in C(B)$ there exists a unique homomorphism of similarity $h : W(B) \rightarrow H(B)$ which coincides with a given function on the generators of $W(B)$.

The HAS-Hierarchy defines a programming language by the ordered triple

$$PL = \langle H(B), W(B), f : H(B) \rightarrow W(B) \rangle$$

where f is obtained from h by an inversion process [8, 33].

Practically each abstract algebra specified by a HAS-Hierarchy consists of a collection of abstract computing types. Each such abstract computing type is a heterogeneous algebra [8, 15, 30] defined by a standard mechanism which consists of the following triple:

Definition Mechanism:

The definition mechanism is given as a finite set of definition patterns that are able to specify an abstract computing type in terms of a collection of abstract computing types already specified. The assumption is that a finite set of predefined abstract computing

types are provided. The common predefined abstract computing types in the conventional languages are **integer, real, char, pointers**, etc.

Declaration Mechanism:

The declaration mechanism is given as a finite set of declaration patterns that allow the instantiation of the actual objects of an abstract computing type specified by the definition mechanism. Most often these objects are constants, variables and operations over a given abstract computing type.

Application Mechanism:

The application mechanism consists of a finite collection of operation schemes that allow us to express computing needs in terms of the objects declared by the declaration mechanism.

The triple $\langle Definition, Declaration, Application \rangle$ has been called the Ada-Spirit [27, 28] since it has firstly been applied in the design of an Ada compiler. At their turn, the computing abstraction types which appear in actual programming languages could be standardized as follows:

1. Abstract Data Types, shortly ADT.
2. Abstract Processing Types, shortly APT.
3. Abstract Control Types, shortly ACT.
4. Abstract Interaction Types, shortly AIT.

The mechanism developed under the name Ada-Spirit together with the above standardization offer a good unified scheme for language specification, language using and language teaching.

Figure 1 representing the basic ideas behind the semantics directed compiler generation can be generalized as in figure 3, providing a natural tool for language specification and implementation (by translation).

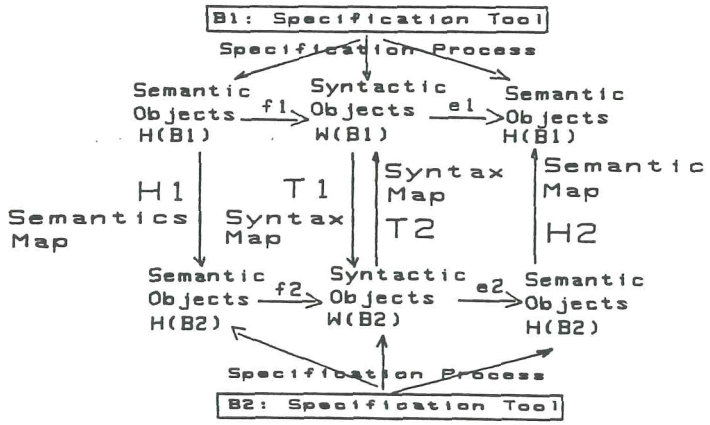

Fig. 3

The mappings f_1, e_1, f_2, e_2, are homomorphisms of similarity. The mappings H_1, H_2, are σ-homomorphisms[15]. The mappings T_1, T_2 are representation homomorphisms [30, 31].

5. TICS Hypotheses

The practical feasibility as well as the efficiency of the theoretical construction of the TICS system are based on some delimitations which result from the human experience in designing and implementing compilers for the conventional programming languages and from the fact that such delimitations do not affect the theoretical power of the approach developed above [26]. Therefore, as hypotheses, we give these delimitations, which in fact are satisfied by all the conventional programming languages and their compilers.

hy1. The specification basis B of a programming language is finite.

hy2. A computing system H(B) is previously given as the language semantics and it has a finite set of generators.

hy3. The specification basis B is provided with a hierarchy relation (or a domination relation) which splits the operation schemes in B into a sequences of classes B_0, B_1, ... , B_n, called the internal hierarchy of the language and denoted by $B_0 \preceq B_1 \preceq ... \preceq B_n$ such that:

a. B_0 is given by the collection of all operation schemes of the arity 0 (zero). That is, B_0 consists of the collection of individual constants provided by B.

b. For each i, i=1, 2, ... , n, we have $B_0 \preceq B_1 \preceq ... \preceq B_i$. The relation $\preceq$ means that B_i dominates hierarchically B_0, B_1, ... ,B_{i-1}, i.e., B_i can be expressed in terms of B_0, B_1, ... , B_{i-1}, or else $B_i = gen(B_0, B_1, ..., B_{i-1})$ where *gen* is the function of generating B_i from the elements of B_0, B_1, ... , B_{i-1}.

In this decomposition n is dependent only on B and is called the hierarchy order of the base B. As a matter of fact *hy3* is given in [30, 31] as a theorem together with the algorithm to construct the hierarchy $B0 \preceq B1 \preceq ... \preceq Bn$.

Example:

In the framework of arithmetic expressions we usually have the following domination relation which expresses the order of operation evaluation:

$$\{\sqrt{},\circ\} \preceq \{*,/\} \preceq \{+,-\}$$

The domination relation stated in *hy3* can be seen as a generalization of the domination relation within the framework of arithmetic expressions.

6. Theoretical Construction

From the specification mechanism discussed in previous section follows that TICS views every program as an expression encapsulating a given computing process. The behavior of this computing process does not depend on its expression as a program. It depends however upon the domination relation $\preceq$ defined on the basis specifying the programming language in which the respective program is expressed.

The first idea behind the TICS mechanism is to describe an universal algorithm which is able to evaluate any given computing process defined in terms of the abstract objects specified by the basis B and represented as a well formed word (i.e. program) in the universal model, i.e. the word algebra generated by B. This is a universal algorithm that computes the values of the function e involved in the language definition. If instead of evaluating the computing process encapsulated into a program this universal algorithm is instructed to evaluate the program expression into another program expression defined as a well formed word of another programming language, then the respective universal algorithm behaves as a translator from the first programming language into the second programming language. In order to develop such an universal algorithm to evaluate a program P by transforming it into another program P', we make the following observations:

Proposition: *According to the relation $\preceq$, the word algebra $W(B)$ can be split on the hierarchical levels $W(B_0)$, $W(B_1)$, ... , $W(B_n)$ which have the following properties:*

1. $W(B_0)$ is finite.
2. For each i, i=1, 2, ... , n, $W(B_i)$ is generated by $W(B_0) \cup W(B_1) \cup ... \cup W(B_{i-1})$ and freely generated by $W(B_0)$.
3. $W(B_n) = W(B)$.

From the properties 1, 2, 3 above the following inclusions result:

$$W(B_0) \subseteq W(B_1) \subseteq ... \subseteq W(B_n) = W(B)$$

The properties 1, 2, 3 are immediate consequences of the results reported in [14, 30, 31]. When they are used to develop an universal algorithm for evaluating a program P, then P must be considered as follows:

1. P is a well formed expression in W(B). Hence, it follows that there exists an i, $0 \leq i \leq n$, such that $P \in W(B_i)$.
2. Let $w_1, w_2, ..., w_k$ be the generators of P in $W(B_0) \cup W(B_1) \cup ... \cup W(B_{i-1})$, i.e., $w_1, w_2, ..., w_k$ are well formed expressions in $W(B_0) \cup W(B_1) \cup ... \cup W(B_{i-1})$ and subwords or subexpressions of P. Then from the definition of the concept of a word (or term) results that P is obtained as a well defined expression in $W(B_i)$ by means of a given operation scheme (or construction) in B_i. This operation scheme can be seen as a triple of the form $\sigma = \langle k; s_0, s_1, ..., s_k; j_1, j_2, ..., j_k, i \rangle$ where $s_0, s_1, ..., s_k$, are fixed symbols, $j_1, j_2, ..., j_k$ are types of $w_1, w_2, ..., w_k$ respectively, $P = s_0 w_1 s_1 ... s_{k-1} w_k s_k$, and i represents the type of P. If $w_1, w_2, ..., w_k$ are already evaluated by the

universal algorithm to the values $\mathcal{V}(w_1)$, $\mathcal{V}(w_2)$, ... , $\mathcal{V}(w_k)$ then the computation operation specified by the operation scheme $\sigma \in B$ and denoted by $F(\sigma)$ computes the value $\mathcal{V}(P)$ in terms of $\mathcal{V}(w_1)$, $\mathcal{V}(w_2)$, ... , $\mathcal{V}(wk)$ as the value

$$\mathcal{V}(P) = F(\sigma)(\mathcal{V}(w_1), \mathcal{V}(w_2), ..., \mathcal{V}(w_n))$$

This is the universal construction in algebra by which a computation process defined on the generators of a given word algebra is extended to the word algebra generated by the generators. When this universal construction is seen as an algorithm, it can be expressed by the flow diagram in figure 4.

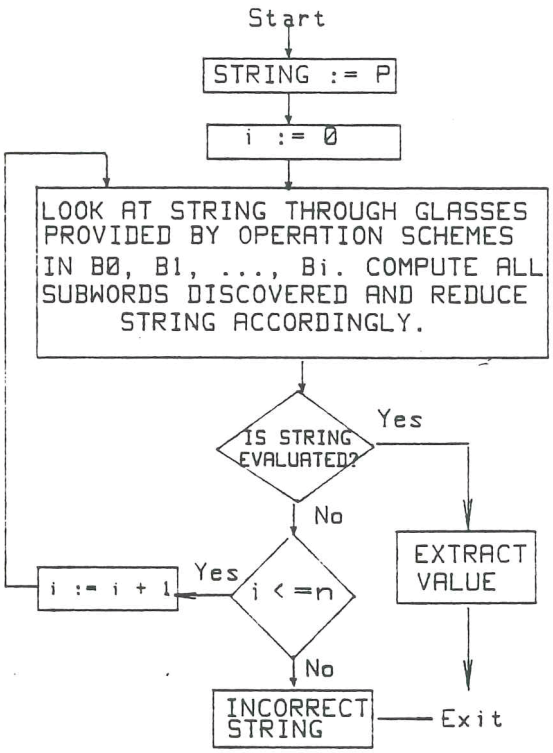

Fig. 4

Now we consider that the computation defined by this universal algorithm is performed into another algebraic structure. For that, a new specification basis B′ is given. Then instead of evaluating P in $H(B)$ according to the computation operation associated by F with the operation schemes in B we assume that the operations associated by F with the operation schemes in B are operating in $W(B')$. This can be achieved as follows:

1. A function $r : B \rightarrow W(B')$ is first defined. This function associates with every

operation scheme in B a well defined word in $W(B')$. Since B is a finite set, this function can be easily constructed by a tabular representation.

2. The relation $\preceq$ defined on B which splits B on the internal hierarchical levels $B_0, B_1, ..., B_n$ is exported by r on $W(B')$, i.e., we have $r(B_0) \preceq r(B_1) \preceq ... \preceq r(B_n)$ where every $r(B_i)$, i=0, 1, ... , n is given by a finite set of words in $W(B')$.

3. For every $\sigma \in B$, $r(\sigma) \in W(B')$ can be considered as a derived operation scheme [18, 22] from the operation schemes in B'. The operations specified by $r(\sigma)$ is denoted by $F(r(\sigma))$. It transports the action of $F(\sigma)$ into $W(B')$.

The following relations hold:

1'. $W(r(B_0))$ is finite.

2'. For each i, i=1, 2, ... , n, $W(r(B_i))$ is generated by $W(r(B_0)) \cup W(r(B_1)) \cup ... \cup W(r(B_{i-1}))$ and freely generated by $W(r(B_0))$.

3'. $W(r(B)) = W(r(B_n))$.

The properties $1', 2', 3'$ tell that the universal algorithm developed by the flow diagram in figure 4 works properly when the computing process is defined by means of the function r above. Therefore, this algorithm can be now organized as an universal algorithm designed to translate well formed words in W(B) into well formed words in $W(B')$ as shown by the flow diagram in figure 5.

When for every $\sigma \in B_0$ we chose in $W(B')$ as the value for $r(\sigma)$ a constant derived operation, i.e., an expression represented in terms of constants in B', then the above algorithm behaves as a compiler from the language specified by B into the language specified by r(B). Since $r(B) \subseteq W(B')$ the compiler is from a language specified by B into a language specified by B'. As a matter of fact this compiler transform W(B) into a subset of $W(B')$, namely W(r(B)).

The domination relation $\preceq'$ defined on B' which ensures the correct behavior of the universal algorithm which evaluates expressions in $W(B')$ in the associated semantics $H(B')$ is hidden in r(B). But because relations $1', 2', 3'$ act on W(r(B)), the relation $\preceq'$ ensures that the universal algorithm acts as the universal construction of extending a computation process defined on the generators of an algebra to the algebra generated by the respective generators. Hence, it follows that the translator defined above preserves the computation process defined in H(B) as the same computation process defined in H(r(B)) which is a substructure of $H(B')$ expressed in terms of $W(B')$.

The algorithm described by means of the flow diagram in Fig. 5 is independent of the program expression and the programming language in which a program is expressed. It depends only on the hierarchical levels of the basis specifying the language in which a program is expressed and on the representation function r . When r is defined as $r : B' \rightarrow W(B)$, a compiler from $W(B')$ into $W(B)$ is obtained.

The actual version of TICS acts on a basis B given by a finite set of BNF rules which are mapped first into operation schemes. These operation schemes satisfy the hypotheses *hy1*, *hy2*, and *hy3*. A representation function r which associates with every operation scheme in B a macro operation in an macro assembly language is used. The particularity

of this association consists of the macro-parameters which are macro-operations. These macro operations are processed at generation time by means of a collection of functions similar to those functions operating in an environment in which a denotational semantics is used. However, they deal with assembly language form of the computing objects instead of their high level semantics expression.

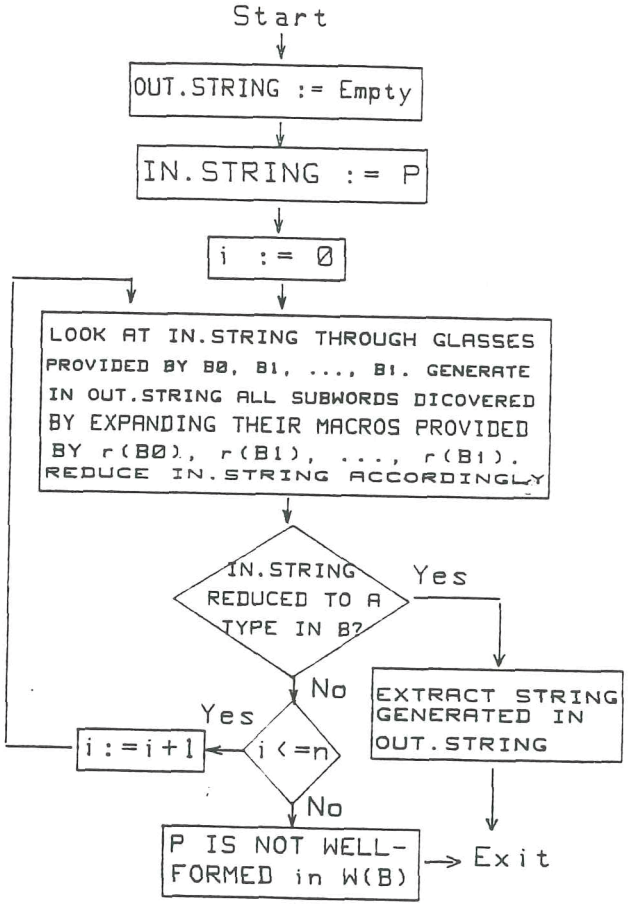

Fig. 5

As shown in [31, 32] for every finite basis B there exists a context-free grammar G(B) such that L(G(B)) = W(B) and for every context-free grammar G there exists a basis B(G) which satisfies the hypotheses *hy1*, *hy2*, *hy3*, and L(G) = W(B(G)). It results that the specification of the actual conventional programming languages and the design and implementation of their compilers can be performed in the framework provided by TICS.

7. Some Practical Solutions

Practically TICS must be seen as a data base whose objects are well defined algorithms represented in some given programming language and operating on standard data structures, organized as tables. Its main components are: a constructor that fed the data structures with appropriate information, a universal compiler acting on the data structure fed by the constructor and a job generator that generates the universal compiler as a program under a given operating system.

In order to implement a new compiler, the constructor receives the source and target language specifications as input data and updates the structure of the universal compiler accordingly. From now on, the universal compiler behaves as a compiler from the source language into the target language received by the constructor. The job generator acts on the updated structure of the universal compiler generating it as a job stream under a given operating system. When this job stream acts on a string, it parses it as an expression of the source language and regenerates it as an expression of the target language. This activity is implemented by TICS as follows:

The constructor:

1. Receives as input data the source and target language specifications given under the form of a finite sequence of rules of the form

$$< A >= "s_0" < A_1 > "s_1" ... "s_{n-1}" < An > "s_n"; MACRO(A)/$$

MACRO(A) represents computing objects of the syntactic category $< A >$ expressed as macro-operations in the target language. A macro assembly language is used for actual implementation. A more general type of macro [34] can be accommodated as well. This sequence of pairs represents in fact the base B and the function r.

2. The constructor splits the specification of the source language into the classes of the hierarchy

$$B_0 \preceq B_1 \preceq \ldots \preceq B_n$$

and updates with these classes the standard tables denoted by TUSO(0), TUSO(1), ... , TUSO(n). The tables TUSO(0), TUSO(1), ... , TUSO(n) accommodate both classes B_0, B_1, ..., B_n and their representations by the associated macro operations denoted by $r(B_0)$, $r(B_1)$, ... , $r(B_n)$.

3. The constructor creates a standard File of the Object Generated by the compiler, FOG, on which the predefined types in B are represented. They are identified by the constructor as specification lines of the form

$$< A >=; MACRO(A)/$$

On FOG they have the form of the assembly code.

The compiler:

The compiler generated by TICS is just the algorithm described by the flow diagram in figure 5, updated to act as follows:

1. TUSO(0), TUSO(1), ... , TUSO(n), FOG, and the standard File of the Internal Form, FIF, are the compiler's input data. FIF represents the internal form of the source program which in figure 5 appears as IN.STRING . A record in FIF is a triple of integers representing the index of the type of the object in the Type Definition Table, TDT, the index of the object itself in the Object Declaration Table, ODT, and the index of the syntax category to which the object belongs. FIF is created by a scanner which represents the first step of the compiler action.
2. The operation LOOK AT IN.STRING is implemented by a fast pattern matching algorithm [17, 32]. It can be optionally implemented by bottom-up parsing algorithm [3, 19, 37] generated from the source language specification.
3. The operation GENERATE IN OUT.STRING is implemented by a macro processor that expands macro operations associated with the syntax patterns discovered by the LOOK AT IN.STRING algorithm. The generation process can be optionally followed by an optimization performed by a local OPTIMIZER. This OPTIMIZER takes as data an assembly section of code generated for a pattern discovered in FIF and optimizes it according to given identities. The result is written in FOG.
4. The operation IN.STRING := P is implemented by a scanner that reads the source text, creates and updates FIF and generates the appropriate portion of the involved tables. The operation EXTRACT GENERATED CODE is implemented by an editor operating on FOG and performing the edition of the target text.

Since the macro-operations in TUSO(0) are associated with non-parameterized macros and from the hierarchical manner of looking at the source string, the bottom-up strategy ensures the following property:

> Every construction in FIF correctly recognized by the compiler can be followed by a successful operation of macro expanding (i.e. target code generation) in FOG because all parameters of the involved macro operation have already been generated in FOG.

FIF and FOG are related by means of the following relation: each object recognized in FIF is generated in FOG and then it is substituted in FIF by the triple $\langle Type, Object, Name \rangle$ where:

1. Type is the index of a record in the Type Definition Table, TDT, in FOG which completely defines the type to which the generated object belongs.
2. Object is the index of the assembly representation of the actual object in the Object Declaration Table, ODT, in FOG. This representation shows the assembly lines composing the object as well as all registers, entry points, exit points, and results used by this assembly language code.

3. Name is the index of the the syntax category of the source language to which the source object discovered by the compiler belongs. A table preserving all symbolic names of syntactic categories is used in this respect.

For efficiency reasons the functions of reading and scanning the original source text has been factored out from the system and an universal scanning algorithm has been provided for it.

An overview of the whole activity is given in figure 6. The rectangle indicates the part of the system which (when updated by the constructor) makes the object of the job generator which in turn generates it as a job stream under a given operating system.

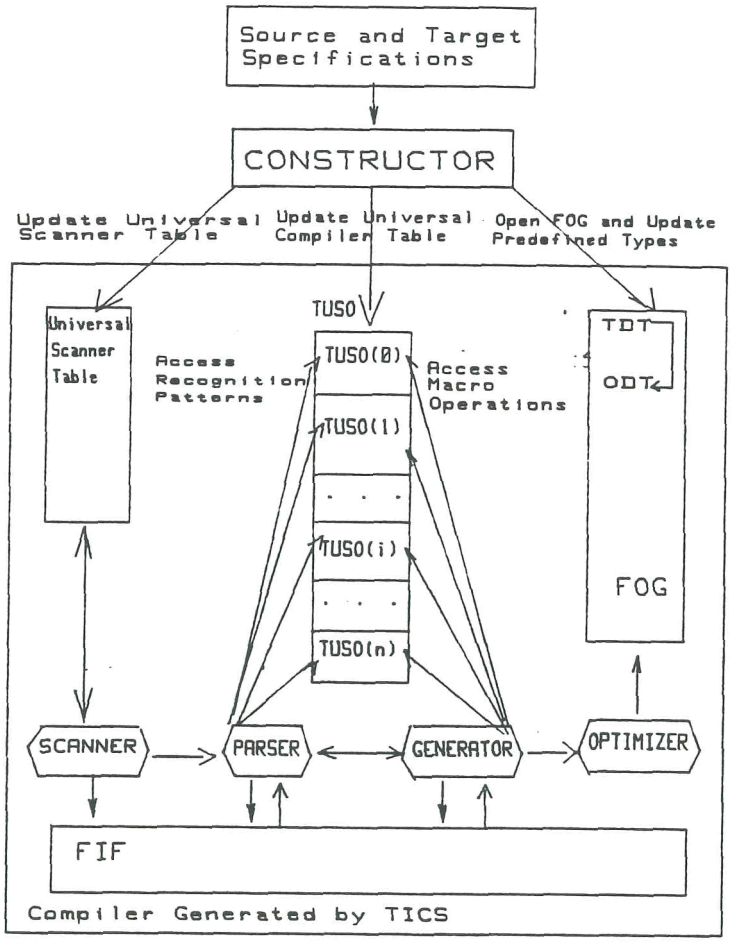

Fig. 6

The basic mechanism to communicate the machine dependent characteristics consists

of the predefined types. They are defined as specification lines of the form

$$< A >=; MACRO(A)/$$

Thus, the constructor can recognize them by their empty right hand side of the BNF rule specifying the language and by the macro-operation associated directly to the type named A. The predefined types together with their operations are organized by the constructor in FOG in the Firmware Definition Table, FDT, shown in figure 7.

Type / Operation	PREDEF_1	PREDEF_2	...	PREDEF_i	...	PREDEF_k
MNEM_1						
MNEM_2						
...	...	...	...	...	...	...
MNEM_j				Actual Code or Pointer		
...	...	...	...	...	...	...
MNEM_n						

Fig. 7

The entry (j,i) in the FDT defines the actual operation represented by the mnemonic MNEM_j on the type PREDEF_i in the macro assembly language used by the compiler. If MNEM_j is an operation defined on the type PREDEF_i then the entry (j,i) shows the machine representation of that operation. Otherwise, a pointer selects a sequence of code representing the actions of MNEM_j on the elements of PREDEF_i in terms of the given machine.

The table FDT offers a general solution for encapsulating in TICS a given machine at the time when it is implemented and to change it very easy. Notice also that FDT allows the compiler implementor to use a small number of mnemonics for macro operation writing. This facility ensures an easy way of handling the target representation and gives a chance to see in fact the whole system as hardware encapsulated.

References

1. ADJ, {J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright}, *Initial Algebra Semantics and Continuous algebras*, Journal ACM **24:1**, 1977, 68 - 95.

2. A. V. Aho, S. C. Johnson, *Optimal Code Generation for Expression Trees*, Journal ACM **23:3**, 1976, 488-501.

3. A. V. Aho, R. Sethi, J. D. Ullman, *COMPILERS Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

4. E. A. Ashcroft, W. W. Wadge, R_x *for Semantics*, ACM TOPLAS **1:2**, 1982, pp. 283-294.

5. R. M. Burstall, J. A. Goguen, *The semantics of CLEAR, A Specification Language*, L N C S: 86, Abstract Software Specification, (G.Goos & J.Harmanis Ed.), 1979, pp.292 - 331.

6. L. M. Chirica, D. F. Martin, *An algebraic Definition of Knuthian Semantics*, Conf. Records IEEE 17-th Annual Symposium on Foundation of Computer Science, 1976, 127 - 136.

7. B. Courcelle, *Attribute Grammars: Theory and Applications*, L N C S: 107, Formalization of Programming Concepts,(G.Goos & J.Hartmanis Ed.), 1981, pp.419 - 431.

8. H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, Springer-Verlag, 1985.

9. H. Ganzinger, *Transforming Denotational Semantics into Particular Attribute Grammars*, L N C S: 94, Semantics Directed Compiler Generation, (G.Goos & J.Hartmanis Ed), 1980, pp.1 - 69.

10. M. C. Gaudel, *Specification of Compilers as Abstract Data Types Representations*, L N C S: 94, Semantics Directed Compiler Generation, (G.Goos & J.Hartmanis Ed.), 1980, pp.104 - 164.

11. M. C. Gaudel, *Compiler Generation from Formal Definition of Programming Languages, A survey*, L N C S: 107, Formalization of Programming Concepts, (G.Goos & J.Hartmanis Ed.), 1981, pp.96 - 114.

12. J. A. Goguen, K. Parsaye-Ghomi, *Algebraic Denotational Semantics Using Parameterized Abstract Module*, L N C S: 107, Formalization of Programming Concepts, (G.Goos & J.Hartmanis, Ed.), 1981, pp.292 - 309.

13. M. J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer Verlag, New York - Heidelberg - Berlin, 1979.

14. W. S. Hatcher, T. Rus, *Context-Free Algebras*, Journal of Cybernetics, **6**, 1976, pp.65 - 77.