to computer-literate and non-computer-literate users alike, and as long as they can provide the capability to perform either completely new tasks, or current tasks faster or at a lower cost than through other media, such as television, newspapers, and the telephone.

In the upcoming years, we should see the further development of multimedia interactive services, as well as the availability of the so-called "Web TVs" and "network computers" which will provide an inexpensive way to access the Internet.

Bibliography

1983. Gecsei, J. *The Architecture of Videotex Systems.* Upper Saddle River, NJ: Prentice Hall.

1997. *IEEE Communications Magazine,* issues on "Residential Broadband Services and Networks", and on "The Global Internet" (June).

Joseph Chammas

VIENNA DEFINITION LANGUAGE

For articles on related subjects *see* BACKUS–NAUR FORM; METALANGUAGE; PROGRAMMING LANGUAGE SEMANTICS; PROGRAMMING LINGUISTICS; and SYNTAX, SEMANTICS, AND PRAGMATICS.

The *Vienna Definition Language* (VDL) is a language for defining the syntax and semantics of programming languages. It consists of a *syntactic metalanguage* for defining the syntax of program and data structures (*q.v.*) and a *semantic metalanguage* that specifies programming language semantics "operationally" in terms of the computations to which programs give rise during execution.

Syntactic structures in VDL may be graphically represented by means of unordered trees (*q.v.*) whose edges are labeled by *selectors.* For example, the expression $a + b$ might be represented in VDL by any one of a set of equivalent unordered trees such as those in Fig. 1.

These tree (t) structures may in turn be represented in linear notation as

$$t = (\langle s_1{:}a\rangle\langle s_2{:}b\rangle, \langle s{-}\mathrm{op}{:}+\rangle)$$

or

$$t = (\langle s_1{:}a\rangle, \langle s{-}\mathrm{op}{:}+\rangle, \langle s_2{:}b\rangle)$$

Selectors in a VDL syntactic structure serve the same role as pointers (*q.v.*) in a list structure and may be used to select components of the syntactic structure by "applying" the selector to the syntactic structure. In the preceding example, $s_1(t)$, $s_2(t)$, $s{-}\mathrm{op}(t)$ yield the respective components a, b, $+$.

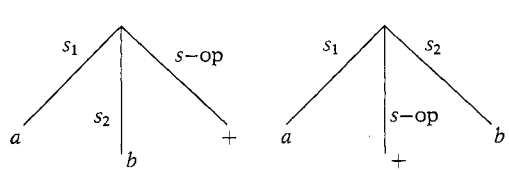

Figure 1.

Syntactic objects may be either *elementary* (*atomic*) *objects* with no components (such as the objects $a, b, +$ above) or *composite objects* (such as the tree t above) whose components may be selected by selectors.

The syntactic metalanguage of VDL is illustrated by the following definition of a simple class of arithmetic expressions:

```
expr = const ∨ var ∨ binary
binary = (⟨s1:expr⟩,⟨s2:expr⟩,⟨s-op:op⟩)
op = {+, *}
```

This definition specifies that an expression can be a *constant* (`const`), a *variable* (`var`), or a *binary*, where constants and variables are elementary objects with no components, and a binary is a composite object with two components of a type "expr" selectable by the selectors s_1, s_2, and a third component of the type "op" selectable by $s{-}\mathrm{op}$. The expression $a + b * c$ may be represented in terms of the preceding syntax by a tree structure whose edges are labeled by selectors as shown in Fig. 2.

If the tree structure in Fig. 2 is denoted by t, then $s_1(t) = a$, $s_2(t) = b * c$, $s{-}\mathrm{op}(t) = +$, $s_1 \cdot s_2(t) = b$, $s_2 \cdot s_2(t) = c$, and $s{-}\mathrm{op} \cdot s_2(t) = *$.

The example illustrates that syntactic objects in VDL are represented by trees whose edges are labeled by selectors, and that components of a tree-structured syntactic object may be selected by specifying the sequence of selectors along the path from the root to the selected subtree.

It is instructive to contrast syntactic specification in VDL with syntactic specification of a corresponding class of expressions in BNF (Backus–Naur form). The previously given class of arithmetic expressions could be specified in BNF as follows:

```
⟨expr⟩ ::= ⟨const⟩|⟨var⟩|⟨binary⟩
⟨binary⟩ ::= ⟨expr⟩⟨op⟩⟨expr⟩
⟨op⟩ ::= +|*
```

The difference between the BNF and VDL syntactic metalanguages is brought out by comparing the two

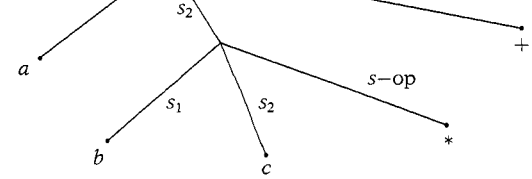

Figure 2.

specifications of `binary`. In BNF a `binary` is a string consisting of an expression followed by an operator followed by a second expression. In VDL a `binary` is a structure with three components selectable by the selectors s_1, s_2, and s–op. If the representation for expressions were changed from infix to prefix notation, so that $a + b * c$ were written as $+a * bc$, then the BNF specification would have to be modified to reflect this change in order, but the VDL representation could remain the same. Because VDL specifies structure independently of the order in which components appear in a specific representation, a VDL syntactic specification is sometimes referred to as an *abstract syntax*.

The *semantics* of a programming language is defined in VDL in terms of the sequences of information-structure transformations to which programs give rise during execution. Every computation starts with an initial configuration ξ_0, which contains a syntactic representation of both the program structure and the data structure on which the program is to operate. Terminating computations consist of a finite sequence of configurations $\xi_0 \rightarrow \xi_1 \rightarrow \cdots \rightarrow \xi_n$, where ξ_{j+1} is obtained from ξ_j by the execution of an instruction. The configurations ξ_j are referred to as *instantaneous descriptions*, *snapshots*, or *states*. The instructions form the heart of the semantic specification of a programming language and have the following general form of definition:

$$\begin{aligned} \text{instruction-name } (x_1, x_2, \ldots, x_n) = p_1 &\rightarrow a_1 \\ p_2 &\rightarrow a_2 \\ &\cdots \\ p_m &\rightarrow a_m \end{aligned}$$

where $p_1, p_2, \ldots, p_m$ are a sequence of predicates, $a_1, a_2, \ldots, a_m$ are a sequence of actions to be performed, and $x_1, x_2, \ldots, x_n$ are a sequence of formal parameters that may appear in the predicate specifications p_i and action specifications a_i.

Example

$$\begin{aligned} \text{abs}(x) = x > 0 &\rightarrow x \\ x = 0 &\rightarrow 0 \\ x < 0 &\rightarrow -x \end{aligned}$$

When an instruction of this form is executed with given actual parameters, the current configuration is tested to see whether it satisfies successive predicates p_i for $i = 1, 2, \ldots, n$. The action a_i corresponding to the first true predicate p_i is then executed. Actions a_i specify transformations of the current configuration ξ_j into the next configuration ξ_{j+1}.

The VDL instruction execution cycle differs from that of conventional computers. At any moment of execution, there is a tree of executable instructions called a *control tree*, and the next executable instruction may be *any* terminal vertex of the control tree. This leads to a certain amount of nondeterminacy in the instruction execution process, which allows VDL to model nondeterminacy in specifying (for example) guarded command (*q.v.*) execution, and also to model nondeterminacy of execution in certain kinds of multitasking (*q.v.*).

There are two kinds of instructions in VDL:

1. Self-replacing instructions, which, when they are executed, replace the terminal vertex of the control tree at which they occur by a subtree of instructions.
2. Value-returning instructions, which return a computed value to predecessor vertices of the control tree and delete the executed instruction from the control tree.

A computation in VDL generally starts with a control tree consisting of a single vertex containing an instruction such as interpret-program (t), where t is the syntactic specification of the program to be executed. The first few executed instructions are generally self-replacing instructions that generate successively larger control trees (determined by the abstract syntax of t) until terminal vertices corresponding to value-returning instructions are generated. Execution terminates when an empty control tree is generated.

The Vienna Definition Language was developed by Peter Lucas, Kurt Walk, and others at the IBM Vienna Laboratory. It has been applied to the definition of PL/I (Lucas and Walk, 1969), Basic (Lee, 1972), and a number of other programming languages. A more detailed introduction to the basic concepts of VDL may be found in Wegner (1972).

Bibliography

1969. Lucas, P., and Walk, K. "On the Formal Description of PL/I," *Annual Review of Automatic Programming*, **6**, *3*, 105–182.

1972. Lee, J. A. N. *Computer Semantics*. New York: Van Nostrand Reinhold.

1972. Wegner, P. "The Vienna Definition Language," *Computing Surveys*, **4**, 5–63.

1997. Harry, A. *Formal Methods Fact File: VDM and Z*. New York: John Wiley.

Peter Wegner

VIRTUAL MEMORY

For articles on related subjects *see* ASSOCIATIVE MEMORY; CACHE MEMORY; DISTRIBUTED SYSTEMS; INTERNET; MEMORY HIERARCHY; MEMORY MANAGEMENT; MEMORY PROTECTION; MULTIPROGRAMMING; OBJECT-ORIENTED PROGRAMMING; OPERATING SYSTEMS; SCHEDULING ALGORITHMS; WORKING SET; AND WORLD WIDE WEB.