Journal of Universal Computer Science, vol. 3, no. 5 (1997), 416-442
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97 © Springer Pub. Co.

# Montages Specifications of Realistic Programming Languages

Philipp W. Kutter
(Eidgenössische Technische Hochschule, Switzerland
kutter@tik.ee.ethz.ch)

Alfonso Pierantonio
(Università di L'Aquila, Italy
alfonso@univaq.it)

**Abstract:** Montages are a new way of describing all aspects of programming languages formally. Such specifications are intelligible for a broad range of people involved in programming language design and use. In order to enhance readability we combine visual and textual elements to yield specifications similar in structure, length, and complexity to those in common language manuals, but with a formal semantics. The formal semantics is based on Gurevich's Abstract State Machines (formerly called Evolving Algebras).

**Key Words:** abstract state machines, language design, Montages, programming languages specifications, visual formalisms, modular specifications

**Category:** F.3.2, D.2.1, D.3.1

## 1 Introduction

In recent years, programming languages followed the trend towards higher levels of abstraction, combining in most cases modularity and object orientedness with simplicity and efficiency. As a consequence, nowadays it is hard to envisage a language which does not support encapsulation in some form. These languages normally required the efforts of large groups of persons and experience suggests that they are not always as compact and coherent as those realized by a single person, as for instance Pascal. Hence, new languages are defined passing through a number of stages, from initial design to routine–use by programmers, forming the so–called programming language life cycle.

During this process, designers need to keep track of already taken decisions and the design intentions must be conveyed to the implementors, and in turn to the users. Therefore, as for other software artifacts, accurate, consistent and intellectually manageable descriptions are needed. So far, the only comprehensive description of a programming language is likely its reference manual, which is mainly informal and open to misinterpretation. Formal approaches are therefore sought. Nevertheless, they are regarded very warily, since they caused a certain dissatisfaction more often than not.

Montages are a semi–visual formalism that allows unified and coherent specification of syntax, static analysis and semantics, and dynamic semantics. They are compositional and easily understandable. The static aspects of Montage descriptions resemble control and data flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. Thus, Montages are a formal instrument which can be equally well understood by language designers, compiler constructors, and programmers.

The semantics of all specification components is formally given using Gurevich's Abstract State Machines (ASMs). ASMs have been successfully used to model the dynamic semantics of Prolog [BR95], Occam [BD96, BDR94], C [GH93], C++ [Wal94], VHDL [BGM95], and Oberon [Kut97]. At the risk of oversimplifying somewhat, we can describe some of these models [GH93, Wal94, Kut97] as follows. Program execution is modeled by the evolution of two functions $CT$ and $S$. $CT$ points to the part of the program text currently in execution and may be seen as an abstract program counter. $S$ represents the current value of the store. Formally one defines the *initial state* of the functions and specifies how they evolve by means of *transition rules.*

These models assume that the representation of the program's control and data flow in the form of functions between parts of the program text is given. The control flow functions specify the order in which statements are executed, and the data flow functions specify how values flow through operations. The transition rules of the models update the program counter and program state using the control and data flow functions. By doing so, the dynamic semantics can be based on the token sequence which is a progress compared to other semantics specifications which are usually based on the abstract syntax tree and assume often that the analysis of static semantics is known. But the lack of formalization of the static analysis remains a heavy limitation to the completeness of the given specification. Moreover, it prevents the specification from being executed using existing ASM–interpreters [DCIG93].

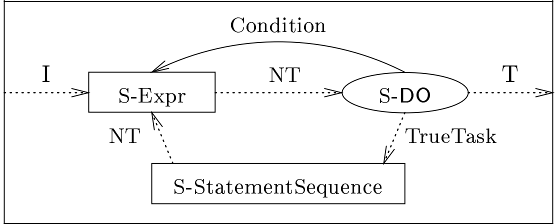
The idea of specifying the static analysis in an ASM setting is not a novelty. The Occam model described in [BD96] accounts for the static analysis and the dynamic semantics, and the work in [MJ94] considered ASM's methods for describing context sensitive formalisms including attribute grammars. Nevertheless, the way Montages express the static analysis is new and provides for a nice formalization of the static semantics as well.

Montages have been used in several case studies. In [KP97] we specify the complete Oberon language in four subsequent refinement steps. Complex features, such as encapsulation, modularity, inheritance, and pointers, are covered in a surprisingly short and comprehensive fashion. Montages is the approach being used at ICSI, Berkeley for the specification of the object–oriented language Sather [Anl97b] using a tool, GEM ([Anl97a]), described in section 4. In [DiF97] the SQL direct (ISO 9075) is formalized.

The paper is organized as follows. In the next section, we first give a tutorial introduction by means of a simple expression language. In section 2.2 the formal semantics of Montages is presented. Section 3 shows some advanced features, which are used in the large case studies. In section 4 we describe the tool support for Montage specifications. Then a comparison with related works is provided in section 5. In the last section we summarize the results and outline some topics for further research.

The Montage specification of a While loop is informally presented. The topmost part is the production rule defining the context–free syntax. Below is a graphical representation of the control and data flow graph. The NT (NextTask), and TrueTask arrows denote, for instance, sequential control flow, while the Condition arrow denotes the data flow. Control flow arrows are dotted and data flow arrows are solid. The control flow arrows I (initial) and T (terminal) are special arrows which serve to plug together the local flow–information to the global one. The boxes and circles denote non–terminal and terminal symbols, respectively. The third part of the While Montage contains the static semantics, that is, the type of the While-condition must be Boolean. The last part contains the dynamic semantics rules. This rule is executed as the abstract program counter (CurrentTask) reaches a DO token. In this case, it checks whether the value of the condition is true. If it is true, the abstract program counter is set to the statement sequence (using the TrueTask arrow), else to the next task. The next task of the DO token is not defined directly by the graph, but it is defined through the mentioned plugging mechanism of the T arrow.

**WhileStatement ::=** **WHILE Expr DO StatementSequence END**




**condition** S-Expr.StaticType = Boolean

DO:

**if** Condition.Value = true **then**
  CurrentTask := TrueTask
**else**
  CurrentTask := NextTask
**endif**

## 2 Montages

A language specification, i.e. the description of all the syntactical and semantical aspects, is given as a collection of Montages, each of which is associated with a production rule. The semantics of such a collection is an ASM. This ASM can be thought as composed of two other ones, the former declaring the static analysis and semantics, the latter defining the dynamic semantics. Each program of the specified language defines an initial state for the ASM, which contains the parse tree. The static analysis decorates the leaves of the tree with the control and data flow building the sequence of token that is needed by the operational semantics. During the static analysis, the static semantics of each node is checked.

### 2.1 Tutorial Introduction

A Montages specification of a language $\mathcal{L}$ defines an ASM $\mathrm{M}_{\mathcal{L}}$ that for a given program $\mathcal{P}$ of $\mathcal{L}$ analyzes and defines the control and data flow, checks the static semantics, and in turn executes the dynamic semantics. The transition rule of $\mathrm{M}_{\mathcal{L}}$ is applied to an initial state $\mathrm{I}_{\mathcal{P}}$, which depends on the program $\mathcal{P}$: for each different program, there is a different initial state.

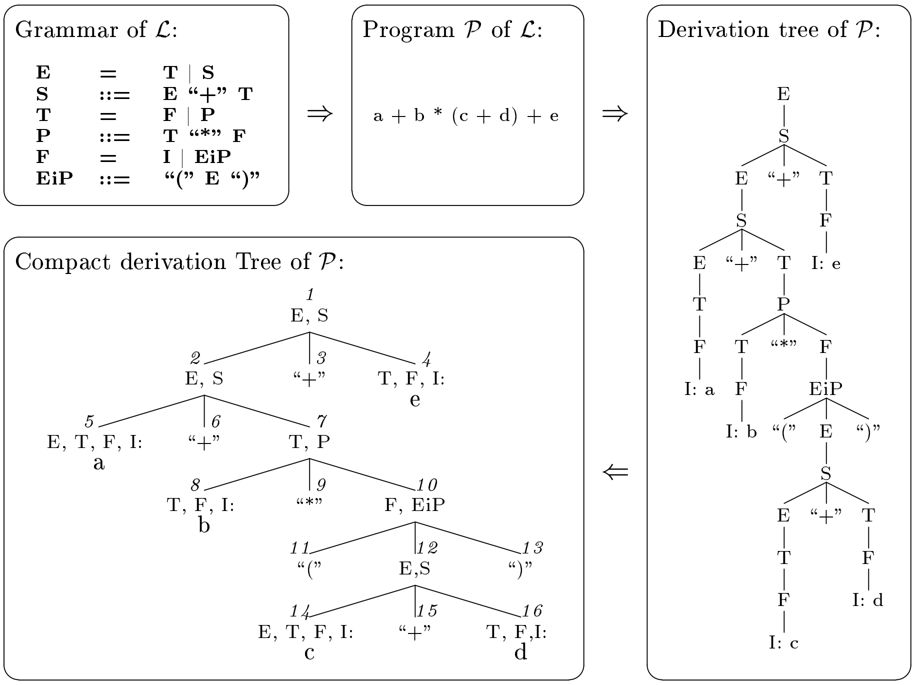



**Figure 1:** Example for a grammar, a program, and its derivation trees

In the sequel, we first illustrate how an initial state is defined based on the syntax of a program, and then how the machine $M_{\mathcal{L}}$ models the statics and the dynamics of the language $\mathcal{L}$.

### 2.1.1 Initial State — Syntax

The initial state $I_{\mathcal{P}}$ encodes the syntax of a program $\mathcal{P}$. $\mathcal{L}$–programs are terms generated by a context free grammar $G_{\mathcal{L}}$. Each program generated by $G_{\mathcal{L}}$ can be represented as a *compact derivation tree*. A compact derivation tree is a parse tree in which a node $n$ together with its single descendant $d$ is collapsed to one node, having both the labels of $n$ and $d$ and having only the descendants of $d$. The initial state $I_{\mathcal{P}}$ associated with a program $\mathcal{P}$ is given by universes and functions representing the nodes and branches of the compact derivation tree. The functions are called *selector functions* since they allow to select the descendants of a node.

Figure 1 contains a context–free grammar of an example expression language $\mathcal{L}$, with non–terminals for *Expression*, *Sum*, *Term*, *Product*, *Factor*, and *ExprIn-Parenthesis*, a term $\mathcal{P}$ of that grammar, the normal derivation tree of $\mathcal{P}$, and its compact version. The sign "=" is used in the grammar instead of "::=" whenever a rule generates only a unique descendant, which can be collapsed with its parent. Such rules are called *synonym productions*. All the following is based on compact derivation trees.

| | |
|---|---|
| S-Expression : *Sum* $\longrightarrow$ *Expression* | $1 \mapsto 2$<br>$2 \mapsto 5$<br>$12 \mapsto 14$ |
| S-Expression : *ExprInParenthesis* $\longrightarrow$ *Expression* | $10 \mapsto 12$ |
| S-Term : *Sum* $\longrightarrow$ *Term* | $1 \mapsto 4$<br>$2 \mapsto 7$<br>$12 \mapsto 16$ |
| S-Term : *Product* $\longrightarrow$ *Term* | $7 \mapsto 8$ |
| S-Factor : *Product* $\longrightarrow$ *Factor* | $7 \mapsto 10$ |
| S-"+" : *Sum* $\longrightarrow$ "+" | $1 \mapsto 3$<br>$2 \mapsto 6$<br>$12 \mapsto 15$ |
| S-"*" : *Product* $\longrightarrow$ "*" | $7 \mapsto 9$ |

**Figure 2:** Selector functions

The initial state $\mathrm{I}_{\mathcal{P}}$ contains the compact derivation tree of the program $\mathcal{P}$. The state $\mathrm{I}_{\mathcal{P}}$ has a universe, called *Node*, which contains both the non–terminal and terminal nodes of the tree. Moreover, we have a sub–universe of *Node* for each non–terminal and terminal symbol, in which are contained the nodes belonging to that category. The universes of the example in figure 1 are the following, where the naturals identify the nodes of the compact derivation tree:

$$\begin{array}{rlrl} \mathit{Expression} = & \{1, 2, 5, 12, 14\} & \mathit{Sum} = & \{1, 2, 12\} \\ \mathit{Term} = & \{4, 5, 7, 8, 14, 16\} & \mathit{Factor} = & \{4, 5, 8, 10, 14, 16\} \\ \mathit{ExprInParenthesis} = & \{10\} & \mathit{Product} = & \{7\} \end{array}$$

$$\mathit{Ident} = \{4, 5, 8, 14, 16\}$$

$$\begin{array}{rlrl} \text{“*”} = & \{9\} & \text{“+”} = & \{6, 3, 15\} \\ \text{“(”} = & \{11\} & \text{“)”} = & \{13\} \end{array}$$

The synonym productions cause some nodes to belong to more than one universe. Together with the universes, in the initial state $\mathrm{I}_{\mathcal{P}}$ there are also the selector functions which link the nodes downwards according to the tree. Since universes are not disjunct these functions are overloaded, i.e. the same selector function is used to select descendants from more than one category of nodes. The definition of the selector functions are given in figure 2. The notation S- stays for "selector" and it is used for distinguishing the function names from the universe names.

### 2.1.2 Static Analysis

The definition of $\mathrm{I}_{\mathcal{P}}$ required only the grammar $\mathrm{G}_{\mathcal{L}}$ of the language $\mathcal{L}$. Now we illustrate how the static analysis of $\mathcal{L}$ is specified using Montages and how the semantics of this specifications generates the control and data flow graph of $\mathcal{P}$.

The control and data flow information is provided as attributes of the tokens. Graphically the attributes can be illustrated as labeled arrows between

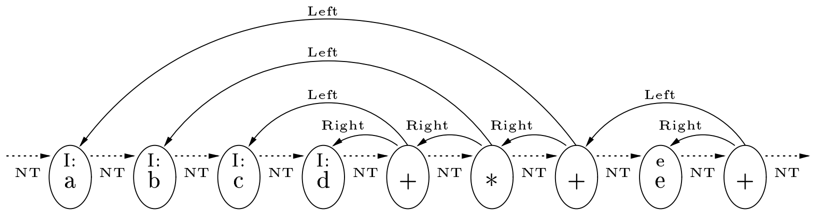



**Figure 3:** Control and data flow of $\mathcal{P}$

the tokens. As convention, we use dotted arrows to denote control flow and solid arrows to denote data flow.

An addition token "+", for instance, has typically two data flow attributes *Left* and *Right* pointing to the tokens representing its left and right arguments. This is illustrated in the control and data flow of $\mathcal{P}$ given in figure 3, which contains the control and data flow of the program $a + b * (c + d) + e$. The NT–labeled control flow arrows define the next task attribute in the control flow. The flow graph links directly tokens of the program text, and does not use internal nodes of the derivation tree.

Montages provides a method for specifying static analysis, i.e. for defining control and data flow starting from the syntax of the language. The parse tree is made of recurrent patterns, since it is generated by a grammar. Each production rule

$$n ::= E$$

has different occurrences in the tree, in particular we can imagine the non–terminal $n$ matching subtrees whose descendants are structured according to the right–hand side $E$ of the production. The root of such a subtree is a node corresponding to the left–hand side $n$. Non–terminal (terminal) symbols in $E$ represent in this view the direct descendants of that root. Linking together this symbols results in flow arrows between internal nodes of the derivation tree. In order to get normal flow graphs we have to move this arrows down to the leaf–level. How to do this is defined by the Montages Method in a simple but universal way, that allows us to define control and data flow graphs by specifying how the symbols in the grammar productions – both terminals and non–terminals – are linked together.

An example are the Montages of $\mathcal{L}$ in figure 4. There we see how the already mentioned arrows *Left*, *Right*, and NT are defined between symbols of the right–hand side of the corresponding (written just above the respective graph) production rule.

A Montages specification consists of descriptions associated with the production rules. Each description, called Montage, contains four parts: the first is the production rule the Montage speaks about; the second contains the graph defining control and data flow as discussed. The third, marked with keyword **condition** contains the static semantics condition, using the flow information (figure 4 does not contain the third part). And the last box contains the dynamic semantics definitions for the tokens introduced by the production rule (in figure 4, Ident, "+", and "*").

Each inner node can be associated with a subtree of the parse tree. The local control flow among the leaves of the subtree is defined by the corresponding

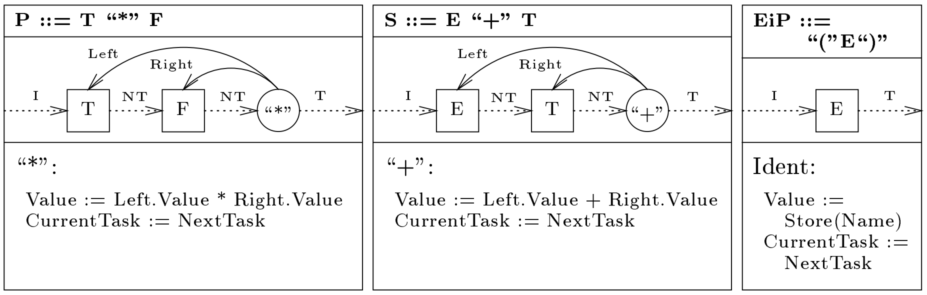



**Figure 4:** Montages of $\mathcal{L}$

Montage. The first and the last leaves of this local control flow are called *initial* and *terminal leaves*. Incoming control flow is always connected to the initial leaf and outgoing to the terminal one. The function Initial connects a node to its initial leaf and the function Terminal to its terminal leaf (see figure 5).

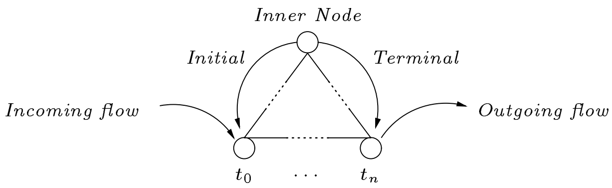



**Figure 5:** Local control flow

The initial and terminal leaves are defined graphically: for each production rule, the static analysis graph contains a dotted arrow from the box border to a symbol, called the *I–symbol* of the production rule, and a dotted arrow from a symbol, called the *T–symbol* of the production rule, to the border of the box (see example in figure 4). We define the initial and terminal functions over each node as follows: they are the identity over the leaves; for each inner node the initial leaf is defined as the initial leaf of its I–descendant and the terminal leaf is the terminal leaf of its T–descendant (see figure 6).

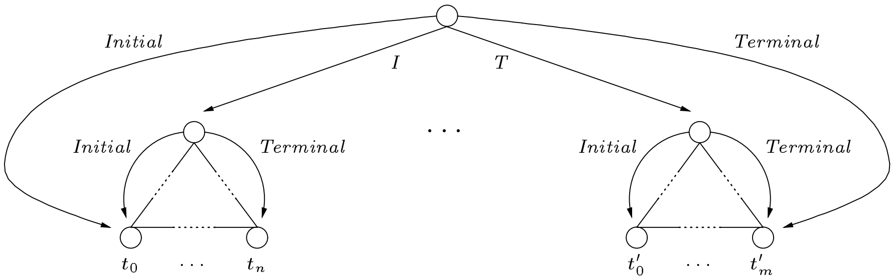



**Figure 6:** Initial and terminal leaves

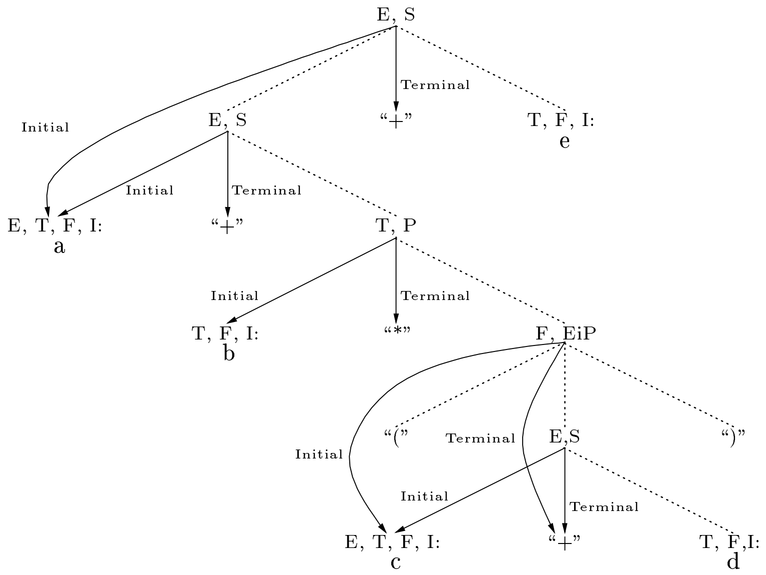



**Figure 7:** Definition of Initial and Terminal in $\mathcal{P}$

Applied to the compact derivation tree in our example, we get the situation in figure 7 where the dotted arrows are the tree structure, and the solid arrows define the initial and the terminal leaves.

Based on the definition of initial and terminal leaf, we can now define which leaves are linked by control and data arrows as follows:

- A *dotted control flow arrow* links the terminal leaf of its source with the initial leaf of its target. This choice is based on the heuristic that control flow typically is some kind of sequential control flow and therefore an arrow from inner node $a$ to inner node $b$ means that after processing the last (terminal) task of $a$, one has to process the first (initial) task of $b$.
- A *solid data flow arrow* links the terminal leaf of its source with the terminal leaf of its target. The rationale is the following: assume, for instance, an inner node representing an expression; the value of the expression can be evaluated, only if all its sub–terms have been evaluated, i.e. they had control. Thus if the control is given to the terminal leaf of the expression, the result is available, and can be stored as field of that leaf. This and other typical situations show, that the terminal leaf is well suited as access point for data flow.

Using these definitions, one sees how the four Montages in figure 4 define the control and data flow graph of figure 3. The start point is the compact derivation tree of $\mathcal{P}$ with the definitions of Initial and Terminal, as given in figure 7. Then for each inner node $n$ in the tree do the following:

1. Find the Montage having one of the labels as left–hand side of its production rule. It is guaranteed that there is only one matching Montage.
2. Define the arrows specified in the graph of that Montage between the descendants of $n$.

3. Change each of these arrows to an arrow between leaves as follows:
   (a) a control arrow from node $a$ to a node $b$ is changed to an arrow from the terminal leaf of $a$ to the initial leaf of $b$;
   (b) a data arrow from node $a$ to a node $b$ is changed to an arrow from the terminal leaf of $a$ to the terminal leaf of $b$.

Up to now we illustrated how to enrich the grammar for specifying the static analysis. Usually Montages have also static semantics constraints, which are first–order predicates. Such predicates are evaluated during the static analysis.

### 2.1.3 Dynamic Semantics

The fourth part of a Montage is the dynamic semantics. It is given by means of ASM's transition rules. A transition rule is a description of how the current state evolves. In figure 4, we have simple rules which describe how the expressions are evaluated.

Given an initial state, the different states of an ASM are reached by iteratively triggering the transition rules until the ASM is in a state which cannot evolve anymore. The initial state of the dynamic semantics is the result of the static analysis. The transition rule characterizing the dynamic semantics is the union of all transition rules given in the lowest part of the Montages.

A sequential program typically runs by executing one *machine instruction* after the other. The current instruction is pointed by a "program counter". We already mentioned that the leaves are considered as the instructions in our model, therefore we call them tasks. As "program counter" we use a nullary function *CurrentTask* (abbreviated CT) pointing always to the task which currently has the control. Typically in each step CT is set to the next task in the control flow, which can be retrieved as the NextTask attribute of the task. In each Montages of $\mathcal{L}$ we find in the dynamic semantics part the update CurrentTask := NextTask which sets the current task to the next task.

Unlike CurrentTask, NextTask is a field (unary function) which takes implicitly the current task as argument if another argument is not indicated explicitly (see section 2.2.3). Each rule is guarded, it is triggered if the CurrentTask points to a token which is labeled by the symbol indicated by the dynamic semantics part of the Montage.

For all types of tasks in $\mathcal{L}$ (e.g. Ident, "+", and "*" tasks) we see that the field *Value* of the current task is updated. If the current task is an Ident, then the value of the store for that variable, accessed by means of the field *Name*, is assigned to the *Value* field

Value := Store(Name)

In this simple example, the store is modeled as a function from variable names to integers. If the current task is an addition (multiplication) operator, then the *Value* field of the current task is set to the sum (product) of the *Value* field of the left task in the data flow and the Value field of the right task in the data flow. The left and right tasks are accessed by the *Left* and *Right* field, respectively (figure 4).

We can now execute the example $\mathcal{P}$ in figure 3. The initial state of the system consists of the definitions of the functions CT, Store and the control and data flow graph. Thus, we need to define the values of the functions CT, and Store. We set CT to the a–task and Store to {(a, 4), (b, 8), (c,3), (d, 7), (e,9)}. The repeated execution of the transition rules will now result in the calculation of 4 + 8 *(3 + 7) + 9. In the first step, the *Value* field of the a–task is set to 4 and parallelly, the current task is set to the b–task. In the second, third, and fourth step, the *Value* field of the b–, c–, and d–tasks are set to 8, 3, and 7. After those steps, the current task is equal to the first "+"–task. According to the transition rule for "+"–tasks, the *Value* field of that task is set to the sum of the *Value* fields of the c– and d–tasks and the current task is set to the "*"–task. This task multiplies the *Value* fields of the first "+"–task and the b–task, and stores the result as its own *Value* field. This process continues, until the last "+"–task sets the current task to undefined, since the NT attribute of the last "+"–task is not defined. At this point the system terminates, no update can be triggered any more. If we made a correct protocol, the *Value* field of the terminal "+"–task holds the result of the calculation: 93

## 2.2 Semantics

In this section we give the formal semantics of Montage specifications. In section 2.2.1 we define how the parse trees are represented in the initial state and describe how the nodes of compact derivation trees are characterized by a specific subset of the symbols in the grammar. This subset, the so–called *characteristic symbols* is the base for the syntax driven modularity of Montages. Section 2.2.2 describes a declarative tree traversal, which is necessary for the static analysis. In section 2.2.3 the Montage notation is illustrated. In the last section, we define the formal semantics of Montages.

We assume the basics of the ASM framework and refer the reader for the definitions in [Gur95].

### 2.2.1 Tree Representation

The generation of a string $S$ by a grammar can be described as usual by means of a *derivation tree*. A derivation tree can be made more compact by putting multiple labels in the case of *synonym productions* [Ode89], i.e. rules of the form $n ::= s_1|s_2|\ldots|s_m$, which give place to nodes with only one child. In such cases we do not append new nodes but we keep track of the synonym productions by adding a new label to the current node. The resulting trees are called *compact derivation trees* and we distinguish a synonym production $n ::= E$ by writing $n = E$.

According to the above definitions, each node is labeled with at least one terminal or one non–terminal, which is the left–hand–side of a non–synonym production. Such symbols are called *characteristic symbols* since it can be shown that each node is labeled with exactly one of them. If a node is labeled with a characteristic symbol $s$ we say as well that the node is characterized by $s$. Such a characterization partitions the set of nodes. This partition is the base for the modularity of Montages.

Given a program, its compact derivation tree is represented in the associated initial state. Our setting requires some specific universes. In particular, the nodes and the leaves of the compact derivation tree constitute the universe *Node* and *Token*, respectively. Moreover, each terminal and non–terminal symbol $s$ is interpreted by a sub–universe of *Node* containing those nodes which are labeled by $s$.

A number of *selector functions* reflects the structure of compact derivation trees and allows us to retrieve the syntactical elements of the program text. Since descendants of a node are constructed by a production rule, we define the functions accordingly. Let $x$ be a node whose descendants have been constructed by a production due to a rule $n ::= E$, then

- If $E$ is of the form "$s_1 s_2 ... s_m$" we access the new nodes in the universes $s_1, s_2 \cdots$, and $s_m$ by unary functions

  $$(\text{S-}s_i : \mathit{Node} \rightarrow s_i)_{i \in \{1,\dots,m\}}$$

  If the same symbol $s$ occurs more than once in "$s_1 s_2 \cdots s_m$", we enumerate the functions from left to right: S1-s maps $x$ to the first $s$–descendant, S2-s to the second and so on.
- If $E$ contains a symbols $s$ in a { } part, then an element of a universe *ListNode* is created and serves as access point of the whole list. The details are given in section 3.2.

We need also an auxiliary function $\mathit{Up} : \mathit{Node} \longrightarrow \mathit{Node}$, which links the descendants to their parents.

In our approach we assumed the initial state to contain the parse tree. The GEM Tool [Anl97a] is a static structure generator generator, i.e. starting from a Montage specification it generates a parser which creates for each program source the initial state for the ASM associated with the language.

### 2.2.2 Tree Traversal

The static parts of a Montage specification describes the static analysis and semantics by transition rules which can be used to define a traversal of the compact derivation tree. Such traversal executes at each node an action, which depends on the characteristic symbol.

We start with a rule executing the action for all nodes in parallel. Then we show how to specify an action depending on the characterization of the node. In order to execute an action R for each node, we use the vary construct of ASMs. The rule

**vary** Self **over** Node (7)
  *R*
**endvary**

executes R for each element in Node simultaneously. The bound variable Self (current node) can be used in R to access the single elements. If for instance Node is a universe with two elements $a$ and $b$, the above rule corresponds to a block of twice R, once with Self substituted by $a$ and once with Self substituted by $b$.

Using the fact that the nodes are partitioned by their characterization, we can execute a specialized rule $\mathrm{R}_n$, the so called *action of n*, for all nodes characterized by $n$ by replacing R in the above vary rule by a block of conditionals, one for each characterizing symbol $n$:

**if** n(Self) **then** (8)
  $R_n$
**endif**

Such a conditional triggers $R_n$ only, if Self is in the universe n. For convenience we say henceforth action of a node, if we mean the action of its characterization.

Up to now, the actions of all nodes are executed in parallel. The next task is to introduce the possibility to sequentialize the execution. Typically the actions should be executed for lower level nodes first, and in some order between the children of a node. The situation where actions of lower level nodes are executed first allows already for direct representation of structural induction: each node (representing a parsed term) can use the results of the actions (definitions) performed for the descendants (representing sub–terms). In addition we need often a certain sequentialization between descendants, e.g. actions for declaration parts in programs must typically be performed before actions for statement parts.

For the sequentialization task, we need a boolean dynamic field

$$\textit{Visited: Node} \rightarrow \textit{Bool}$$

which is initialized with *false* for each node. This field indicates whether a node has been visited, i.e. whether its action has been executed. A relation

$$\textit{before: Node} \times \textit{Node} \rightarrow \textit{Bool}$$

relates nodes sequentially. The relation ($a$ *before* $b$) indicates that node $a$ must be visited before node $b$. The relation *before* can be defined with a parallel tree traversal (see the next section).

Using the above definitions, a sequentialized traversal is defined by the following rule

**vary** Self **over** Node (9)
  **satisfying**
    **for all** node **in** Node **holds**
      node before Self **implies** node.Visited
  $R$
  Self.Visited := true
**endvary**

where again R is refined to a case distinction by characterization (8). The final state or termination of a sequentialized tree traversal is typically reached if the root of the tree is visited.

### 2.2.3 Notational Shortcuts

The textual parts of the Montage specification adopt some syntactical conventions in order to enhance the readability while retaining a full mathematical rigor. The conventions are based on the fact that there are four kindes of functions in Montage specifications:

- *selector functions* These functions are statically defined by the representation of the parse tree and they are marked by the S- prefix.

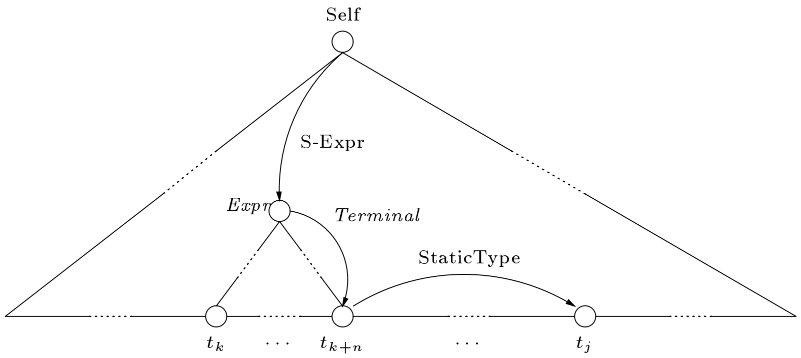



**Figure 8:** Explanation of S-Expr.StaticType

- *initial and terminal* As discussed these two functions link the inner nodes of the parse tree with the tokens.
- *field functions* These dynamic functions take always a token as their first argument and deliver another token or some value. They can be specified graphically by means of arrows.
- *global functions* These functions are used for global links, such as a symbol table in the static analysis.

The notational shortcuts concern terms built up only from selector functions and field functions.

(a) *static terms* Such terms are used in the static part of a Montage specification, e.g. static analysis and static semantics conditions. Their syntax is a possibly empty list of selector functions followed by a possibly empty list of field functions. As usual the functions are composed by means of a ".". If both lists are empty then Self is used. For instance, a valid static term is the one in the condition of the While Montage (box on page 3), i.e.

S-Expr.StaticType (10)

This is a shorthand for ASM terms used within the tree traversal pattern (3). In fact, the above syntax is translated in the ASM notation by prefixing the term with Self and inserting between the selector functions and the fields the function Terminal. The static term (10) is therefore translated to

Self.S-Expr.Terminal.StaticType

The explanation can be found in the way the tree is represented and traversed, this is depicted in figure 8. The selector functions are used for defining a path in the parse tree, the terminal maps the term to the leaves level, and the field functions define a path among the tokens. Consider one may have a nesting of both selector and field functions.

(b) *dynamic terms* We define the dynamic terms as those terms which are used in the dynamic part of a Montage and which are built up only from field functions. Fields either store data or provide a link to another token. The starting point for the application of field functions is always the current task. Thus, we do not write it explicitly, and a dynamic term is a list of field functions. For instance, the term

$$\text{Condition.Value} \tag{11}$$

in the conditional rule in the While dynamics is translated into the following term

$$\text{CurrentTask.Condition.Value}$$

The dynamic rule in each Montage performs a case distinction on the token universe. Such a rule has always the following form

**if** TokenUniverse(CurrentTask) **then** (12)
  $R$
**endif**

The rule $R$ is triggered each time the nullary function CurrentTask points to a token belonging to the universe TokenUniverse. In order to make the presentation of the dynamic rule less verbose we use the following convention to represent the rule (12)

TokenUniverse:
  $R$

It is a mere syntactic sugar, but together with the static and dynamic term syntax, it improves considerably the presentation of the overall specification.

The syntactical translation of the static and dynamic terms should be performed after the application of the user defined macros. Clashes between global functions and fields are solved in the context or by explicit mentioning them.

#### 2.2.4 Semantics of Montages

The semantics of a Montages specification is an ASM $\mathcal{M}$ that for a given program checks the static semantics, initializes the control and data flow functions, and in a second phase executes the dynamic semantics. The transition rule of $\mathcal{M}$ consists thus of two rules, one modeling the first phase, called *statics rule*, and one modeling the second phase, called *dynamics rule*.

The statics rule is a sequentialized traversal (3). The visual part and the condition of a Montage of a symbol $s$ define the action of $s$ in this traversal. The definition of the *before*–relation for the statics rule is done by a parallel traversal (1) with case distinction by characterization (2). The actions of this traversal are defined such that lower nodes in the tree must be visited before higher nodes, and that siblings are visited in the order corresponding to their left–before–right and top–before–bottom order in the graph of the Montages.

We can thus define the $s$-action in the parallel traversal defining the *before*-relation of the statics rule as follows:

before(Self.$S_1$, Self.$S_2$) := true
before(Self.$S_2$, Self.$S_3$) := true
...
before(Self.$S_{n-1}$, Self.$S_n$) := true
before(Self.$S_n$, Self) := true

where $S_1$, $S_2$, $\ldots$, $S_n$ are the selector functions accessing the descendants of an $s$-node in the left-before-right and top-before-bottom order defined by the Montage of $s$.

*Example* The corresponding action for the While Montage (box on page 3) is

before(Self.S-Expr, Self.S-DO) := true
before(Self.S-DO, Self.S-StatementSequence) := true
before(Self.S-StatementSequence, Self) := true

The actions of the statics rule are explained step by step in the following. Lets assume for the discussion a fixed Montage for a symbol $s$. The action for this Montage is built up as block of updates. Each arrow in the control and data flow graph defines one update in the action. This update links not directly the graphically related nodes but two of their leaf-descendants, in particular those given by the functions

$$\mathit{Initial} : \mathit{Node} \longrightarrow \mathit{Node} \qquad \mathit{Terminal} : \mathit{Node} \longrightarrow \mathit{Node}$$

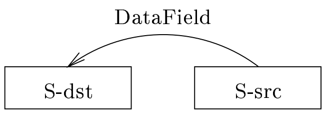
which denote the first and the last leaf in the control flow between the leaf-descendants of a node.

The arrows in the graph define three different kind of updates in the action, one for the above described Initial and Terminal functions, one for data flow arrows,and one for control flow arrows:

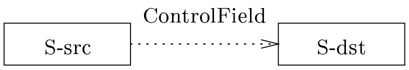
1. To define the functions *Initial* and *Terminal*, we specify which node in the graph contains the initial and terminal leaf, respectively. We call this nodes I–node and T–node. I–node is defined as the target of a dotted arrow labeled with I, and T–node is the source of a dotted arrow labeled with T. The corresponding fragment of transition rule obtained by specifying I–node and T–node graphically is the following block:

   Self.Initial := Self.S-I.Initial
   Self.Terminal := Self.S-T.Terminal

   where S-I is the selector function linking the parent node, corresponding to Self, with its I–descendant and S-T is the selector function linking the parent node with its T–descendant. We call this *fragment–1.*
2. Each solid edge as, for instance, the following

   defines the update Self.S-src.Terminal.DataField := Self.S-dst.Terminal which links the terminal leaves of the source and the target. Such arrows, called *data flow functions* are allowed between square and circle nodes. We call the block of these updates *fragment–2.*
3. Each dotted edge as the following




defines the update Self.S-src.Terminal.ControlField := Self.S-dst.Initial which links the terminal leaf of the source with the initial leaf of the target. Such arrows, called *control flow functions*, are as well allowed between square and circle nodes. We call the block of these updates *fragment–3.*

The action of the statics rules may contain in addition to the updates corresponding to the arrows a rule which is given textually in the second part of the Montage (see section 3). The form of this rule resembles that of updates generated by the second fragment, using the notational conventions.

The static semantics condition is checked before the updates of the action happen, and a nullary function *Abort* is set to true if the condition is false. In order to make the rule easier to read, we write the corresponding conditional rule at the beginning of all updates. The condition predicate is also written using the notational conventions, in particular the ones for the static terms.

The action of a characteristic symbol $s$ in the sequentialized traversal being the statics rule of the Montages semantics is

**if not** Condition **then**
  Abort := true
**endif**

| |
|---|
| *fragment–1.* |
| *fragment–2.* |
| *fragment–3.* |

TransRule

where Condition is the static semantics constraint of Montage $s$, TransRule is the textual rule in the second part of Montage $s$, and the *fragment–1.*, *fragment–2.*, and *fragment–3.* are the updates defined by the graph of Montage $s$. The Condition and TransRule being written using the notational conventions need to be syntactically resolved.

*Example* We give for the While (box on page 3) Montages the corresponding action, as follows:

**if not** Self.S-Expr.StaticType = Boolean **then**
  Abort := true
**endif**

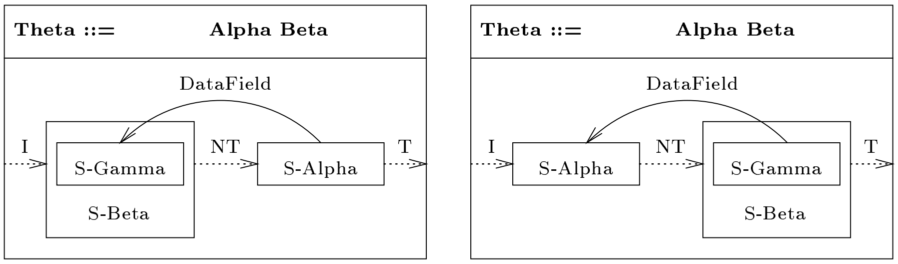
| | |
|---|---|
| Self.Initial := Self.S-Expr.Initial<br>Self.Terminal := Self.S-DO.Terminal | *fragment-1* |
| Self.S-DO.Terminal.Condition := Self.S-Expr.Terminal | *fragment-2* |
| Self.S-Expr.Terminal.NT := Self.S-DO.Initial<br>Self.S-DO.Terminal.TrueTask := Self.S-StatementSequence.Initial<br>Self.S-StatementSequence.Terminal.NT := Self.S-Expr.Initial | *fragment-3* |

## 3 Advanced Features

In the previous section we gave the basics of the Montage visual language. Several aspects of language specifications must deal with scaling up to realistic languages. It is important to avoid the scattering of the knowledge among the specification which would cause a combinatorial explosion. Therefore, in section 3.1 we introduce a technique which allows to specify within one Montage control and data flow which relates nodes produced by two different production rules. Such arrows are called inter–level arrows. Section 3.2 presents a convenient aid to list processing. Both these features enhance expressiveness without compromising the readability and the manageability of the overall specification (see [KP97]). In section 3.3, we sketch how parallel and non–deterministic evaluation can be modeled with Montages.

### 3.1 Inter–level arrows

In the previous section, we have seen arrows between nodes and their meaning in terms of ASM. Control and data flow may require more complex arrows than just those relating siblings. The inter–level arrows are still arrows between boxes but they refer to boxes which are inside other ones (see figure 9 and figure 10).




(a) (b)

**Figure 9:** Samples of inter–level arrows

A box inside another one is a constraint over the non–terminal represented by the outer box. It requires that the rule associated to such non–terminal, or to its synonyms, must contain the symbol associated with the inner box. Figure 9 shows two of such inter–level arrows, in both cases the rules associated with the non–terminal Beta must have the non–terminal Gamma in the right-hand–side, therefore they can be either Beta ::= $\cdots$ Gamma $\cdots$ or Beta = Beta$'$ | Beta$''$, Beta$'$ ::= $\cdots$ Gamma $\cdots$, and Beta$''$ ::= $\cdots$ Gamma $\cdots$ In the examples, the inner boxes are square boxes, but they might have been circle boxes as well. The arrows in figure 9.a and 9.b are translated in the following two rules, respectively

Self.S-Alpha.Terminal.DataField := Self.S-Beta.S-Gamma.Terminal
Self.S-Beta.S-Gamma.Terminal.DataField := Self.S-Alpha.Terminal

A more complex example of inter–level arrow is in figure 10, where an arrow accesses a box which is nested within a number of boxes. The situation is not

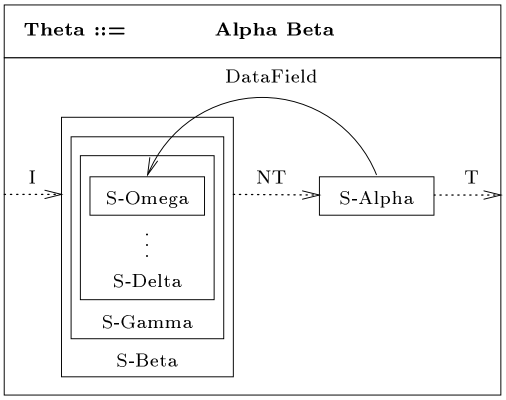



**Figure 10:** Yet another inter–level arrow

more complex than in the previous cases. The nesting of boxes requires that the grammar rule(s) associated to Beta contains the non–terminal Gamma, whose rule(s), in turn, containes Delta, and so on until Omega. This is translated to the following

Self.S-Alpha.Terminal.DataField :=
  Self.S-Beta.S-Gamma.S-Delta. $\cdots$ .S-Omega.Terminal

The most general case in which a data flow arrow, say DataField, goes from a box S-$s$ within the nesting S-$s_1$, ... ,S-$s_n$ to a box S-$s'$ within another nesting S-$s'_1$, ... , S-$s'_m$, requires that the corresponding grammar rules contain all the symbols which appear in the two nestings. The arrow is translated in the following update

Self.S-$s_1$. $\cdots$ .S-$s_n$.S-$s$.Terminal.DataField :=
  Self.S-$s'_1$. $\cdots$ .S-$s'_m$.S-$s'$.Terminal

Inter–level arrows defining control flow are defined in a similar way. If we consider the most general case as the one above, the translation is as follows

Self.S-$s_1$. $\cdots$ .S-$s_n$.S-$s$.Terminal.ControlField :=
  Self.S-$s'_1$. $\cdots$ .S-$s'_m$.S-$s'$.Initial

Inter–level arrows are very useful, especially when combined with the specification of lists, as we will see in the next section.

### 3.2 List Processing

In many approaches a major part of a language specification is concerned with the processing of lists. Therefore we decided to include in Montages a simple, yet powerful list model together with graphical and textual specification elements that can be used to avoid explicit list processing.

If the right–hand–side of a production rule contains a symbol in a { } part, a list of descendents is generated. An additional node, a so–called *list node*, is

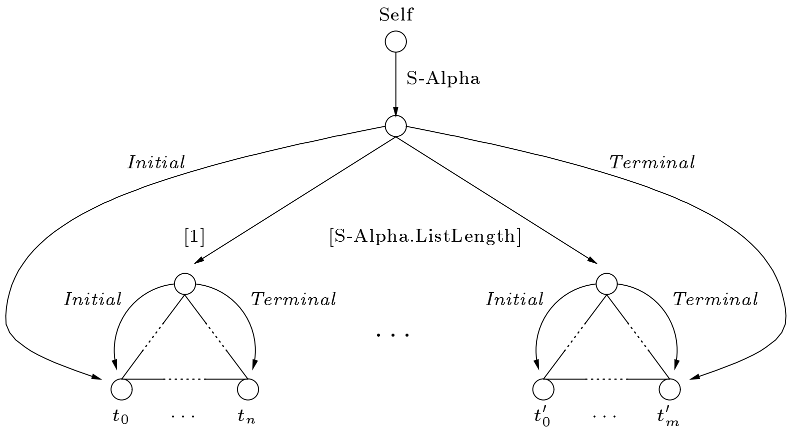



**Figure 11:** The tree associated with a list node

generated as well. It provides access to the elements and to all needed informations about the list. An attribute *ListLength* of the list node is set to the length of the generated list and a binary infix function

$$\_[\_] : ListNode \times Nat \rightarrow Node$$

can be used to retrieve the elements of the list. Moreover, a function

$$Position : Node \longrightarrow Nat$$

returns the physical position of an element within a list. The initial and terminal leaves of a list node are defined to be the initial leaf of the first element, respectively the terminal leaf of the last element in the list. If the list is empty, they point to the list node itself, which then serves as dummy element. The dynamic semantics of that dummy element corresponds to the skip command.

For convenience we assume that a number of patterns in the right-hand-side of production rules are recognized and treated as simple lists. These patterns are

{s}   s{s}   s {aTerminal s}   [s {aTerminal s}]

where aTerminal is usually a separator and $s$ a symbol of the grammar. For all these patterns just one list node is generated, which can be accessed by the selector function S-s. The _[_] function can then be used to access all generated $s$-descendants from left to right, regardless of which $s$ in the pattern generated the descendant. The situation is depicted in figure 11.

A typical use of lists is, for instance, a variable declaration. Figure 12 describes a Montage, whose production rule generates a list of Var–descendants and a node labeled with Type. The list is graphically represented by a box, which is labeled in the upper right corner with the keyword LIST. The single *StaticType*–arrow specifies a family of data flow arrows, one from each variable object to the type–node. At the same time, the action of the list node links all variable objects sequentially with a NextTask arrow.

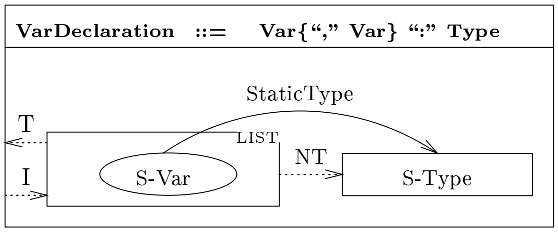



**Figure 12:** A variable declaration Montage

The section is organized as follows. In section 3.2.1 we present the static analysis for list nodes. Section 3.2.2 defines the semantics of inter–level arrows involving list boxes.

### 3.2.1 Static Analysis

We introduce now the static analysis for the list nodes. For the sake of simplicity, we define the action in the *before*-definition such that the elements must be visited from left to right and that they are visited before the list node Self.

$$
\begin{array}{l}
\textbf{vary } i \textbf{ over } \{1, \ldots, \text{Self.ListLength - 1}\} \\
\quad \text{before}(\text{Self}[i], \text{Self}[i+1]) \\
\textbf{endvary} \\
\text{before}(\text{Self}[\text{Self.ListLength}], \text{Self}) := \text{true}
\end{array}
\tag{13}
$$

In addition, the action in the statics rule of a list node always links the elements with a NextTask control arrows and sets the initial and terminal leaf to the corresponding leaves of the first and last element:

$$
\begin{array}{l}
\text{Self.Initial} := \text{Self}[1].\text{Initial} \\
\text{Self.Terminal} := \text{Self}[\text{Self.ListLength}].\text{Terminal} \\
\textbf{vary } i \textbf{ over } \{1, \ldots, \text{Self.ListLength}\} \\
\quad \text{Self}[i].\text{Terminal.NextTask} := \text{Self}[i+1].\text{Initial} \\
\textbf{endvary}
\end{array}
\tag{14}
$$

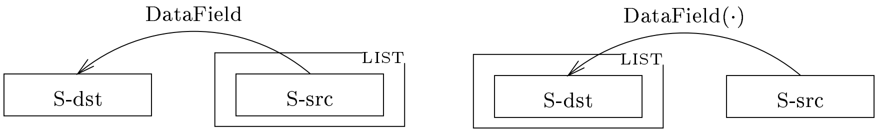
In general, we do not need to fix an *a priori* ordering valid for all the lists as we are doing. One possibility would be to choose among a number of ordering policies which may be indicated in the statics graphs as well. Moreover not necessarily the control flow order defined by the NextTask function in (14), must be the same as the static one in (13). Nonetheless, the case–study in [KP97] showed that this solution works fine with the most complex cases we had to cope with in the specification of Oberon.

### 3.2.2 Semantics

As mentioned, lists have a graphic counterpart. A square or a circle box within a list box is the generic representant of the elements in the list (see figure 13). List arrows, i.e. arrows which involve list boxes, may access to nodes within list boxes or may depart from them, as well. In the rest of the section, we give the semantics of such arrows.


(a) (b)

**Figure 13:** List boxes




(a) (b)

**Figure 14:** List arrows

An arrow from a node within a list box as in figure 14.a corresponds to a family of arrows from all the elements in that list. Formally it defines the following update

**vary** $i$ **over** $\{1, \ldots, \text{Self.S-src.ListLength}\}$ (15)
  Self.S-Src $[i]$.Terminal.DataField := Self.S-dst.Terminal
**endvary**

in other words, it results in a multiple definition of the function DataField over the terminal leaves of the item in the list S-src. The rule (15) can be abbreviated as

**vary** $x$ **over list** Self.S-src
  $x$.DataField := Self.S-dst.Terminal
**endvary**

where **list** $l$ denotes the set of all elements in the list of list–node $l$, as follows

**list** l $\triangleq$ $\{\text{l[i]} \mid \text{i} \in \{1, \ldots, \text{l.ListLength}\}$

The situation changes considerably if we consider the list arrow in figure 14.b. In this case, the function should point to all the items of the list. Therefore, the field DataField is a binary function

$$DataField: Token \times Nat \longrightarrow Token$$

which links a leaf to all the items in the list and the natural denotes the position of the item in the list. The semantics of the arrow is the following

**vary** $x$ **over list** Self.S-dst
  Self.S-src.Terminal.DataField($x$.Position) := $x$.Terminal
  $x$.Terminal.DataFieldPosition := $x$.Position
**endvary**

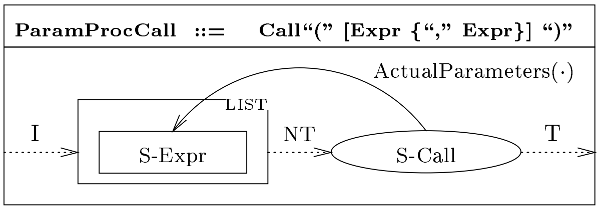



**Figure 15:** A Montage for a procedure call

where DataFieldPosition is the position function. Its name has been obtained by postfixing the field name, i.e. DataField in this case, with Position. Each arrow determines a different position function. The reason is that the position of an item within a nesting of list boxes may be relative to the source of the arrow which points to it.

An example of this kind of list arrow is, for instance, needed for the actual parameters of a procedure call, which is in figure 15. The ActualParameters arrow in the Montage defines a binary function ActualParameters which maps a call task $c$ and a position $n$ to the n–th actual parameter of $c$.

Nested lists are common in programming languages. They can be given a general semantics. We have already seen that nested boxes correspond to a composition of the corresponding selector functions. This holds also for the list boxes with some minor adjustments.

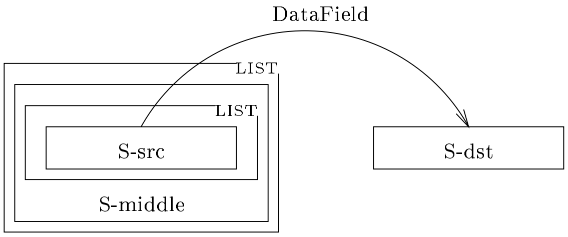



**Figure 16:** A nested list arrow

An arrow which goes from a box within two nested list boxes defines a family of family of updates. For instance, the arrow in figure 16 defines the function DataField in, say, two degrees of freedom, since its semantics is the following

**vary** $x$ **over list** Self.S-middle
  **vary** $y$ **over list** $x$.S-src
    $y$.Terminal.DataField := Self.S-dst.Terminal
  **endvary**
**endvary**

Figure 17 shows two arrows which both reach a box within two nested list boxes. Similarly to figure 14.b, we have a function DataField1 which points to all the elements of the nested lists, therefore it is a function of arity three

$$\mathit{DataField1} : \mathit{Token} \times \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Token}$$

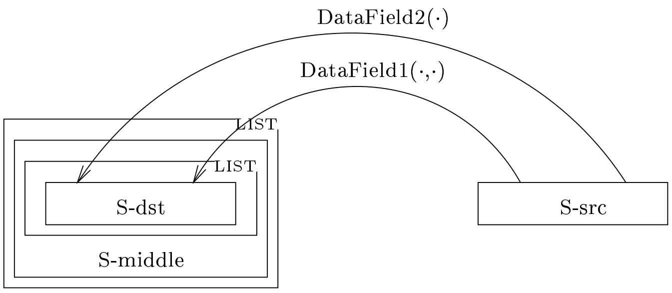



**Figure 17:** Yet other list arrows

The position of each item is given by a pair of projection functions

$$\begin{array}{ll} \textit{DataField1Position1} : \textit{Token} \longrightarrow \textit{Nat} & \text{and} \\ \textit{DataField1Position2} : \textit{Token} \longrightarrow \textit{Nat}, & \end{array}$$

respectively. The semantics is the following

**vary** $x$ **over** Self.S-middle
  **vary** $y$ **over** $x$.S-dst
    Self.S-scr.Terminal.DataField1($x$.Position,$y$.Position) :=
        $y$.Terminal
    $y$.Terminal.DataField1Position1 := $x$.Position
    $y$.Terminal.DataField1Position2 := $y$.Position
  **endvary**
**endvary**

In other words, the two lists are viewed as forming a matrix and the position functions return the position of a given item in the matrix.

Although this is a very natural way of conceiving a nesting of lists, experience suggested that often one may prefer other ways of arranging the lists. In particular, it results too verbose to do quantifications over each list. Therefore, the field DataField2 accesses the items in a linear way, as if they were belonging to a single list, allowing to quantify the items in just one dimension. The semantics of DataField2 is

**vary** $x$ **over** Self.S-middle
  **vary** $y$ **over** $x$.S-dst
    Self.S-src.Terminal.DataField2(Linear($x$.Position,$y$.Position)) := $y$.Terminal
    $y$.Terminal.DataField2Position := Linear($x$.Position,$y$.Position)
  **endvary**
**endvary**

where Linear is a macro defined as follows

Linear($u$,$v$) $\triangleq$
  $\sum_{k=1}^{u\text{-}1}$ Self.S-middle [$k$].S-dst.ListLength + $v$

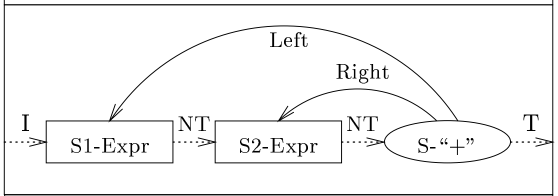
### 3.3 Parallel and Non–deterministic Evaluation Order

In most programming languages, the order in which arguments of expressions are evaluated is not defined. In some, e.g. ANSI–C, it is even not defined that the arguments must be evaluated sequentially. A specification should thus be abstract and not fix the evaluation order. In the Montages framework this situation can be modeled using the same technique as for the sequentialized graph traversal (3): all tasks of a program are executed in parallel, and a relation is used to sequentialize the execution partially.

In order to do so, the initial leaf must be generalized to a set of initial leaves. The semantics of a control flow arrow defines then a relation rather than a function. For each control arrow this relation is established from the single terminal leaf of the source to each initial leaf of the target. Instead of the abstract program counter CurrentTask, there is a set *ToDo* of tasks which must still be executed. Non–determinism can be modeled by an external function which prevents certain tasks in the ToDo set from being executed immediately. The dynamic behavior of this function determine whether all arguments are evaluated at once, in some sequential order, or in a mixed form, e.g. not allowing more than a certain number of additions to be done in parallel.

Multiple initial leaves can be specified graphically by several I–arrows. The semantics of a Montage with several I–arrows is that the initial leaves are the union of all initial leaves of all descendants pointed by an I–arrow. The terminal leaf remains the terminal leaf of the descendant pointed by the T–arrow. As example we show a sequential Sum Montage and then we give a parallel version of it. Figure 18 shows the sequential one. The static semantics of the sum expresses

**Sum ::= Expr "+" Expr**




**condition** Is_Num(S1-Expr.StaticType)
**and** Is_Num(S2-Expr.StaticType)

"+":

Value := Left.Value + Right.Value
CT := NextTask

**Figure 18:** A Montage for a Sum expression

that both components must be of numeric type. In the definition we use a static function Is_Num(·) which maps all numeric types to true. The graph specifying control and data flow defines again NextTask control flow arrows, and field arrows Left, Right, which are used to reference the left respectively right argument of the "+" token. The dynamic semantics rule of the "+"–token assigns to the Field Value of the CurrentTask the result of the addition and control is passed to the next task. In figure 19 there is the Montage of the parallel Sum. Apart

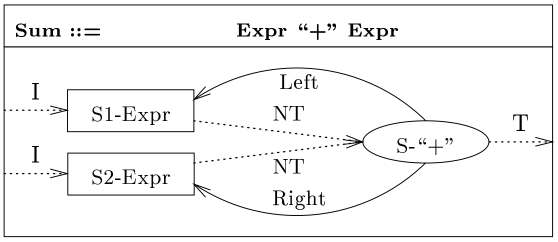



**Figure 19:** A Sum Montages where the arguments may be evaluated in parallel

from the statics graph the other parts of the Montage are identical with the sequential version. Detailed solutions to the problems related with parallelism, e.g. concurrent recursive procedure calls, and a discussion why it is important to abstract from a concrete evaluation order are given in [KH97].

## 4 Tool Support

At the moment, different endeavors are being carried out to provide Montages with an adequate tool support. The GEM Tool[1] has been implemented at ICSI in the Sather project. It is a stand–alone application and consists of a graphic front–end which assists the designer in editing and managing the specification. From the specification it generates a static structure generator, i.e. a parser able to build the initial states for the ASM defined by the Montage specification. High–quality hyper-textual presentation of the specification can also be generated automatically.

Another tool is being realized at INRIA in Sophia–Antipolis [DGKP97]. By using the symbolic representation of the specification generated by GEM, it realizes the static part of Montages using Centaur. This tool is not an implementation but executes a natural semantics version of the Montage's formal semantics.

## 5 Related Work

We have used Kahn's Natural Semantics [Kah87] for the dynamic semantics of Oberon [Kut96]. Although we succeeded due to the excellent tool support by Centaur [BCD$^+$87], the result was much longer and more complex then the ASM counterpart given in [Kut97], since one has to carry around all the state information in the case of Natural Semantics. Natural Semantics and many other settings have the problem of the scattering of the knowledge, i.e. the specification of a construct does not refer to a local piece of formal description, but rather to the whole specification causing a certain combinatorial explosion.

The major similarity between attribute grammars [Knu68] and Montages is that both have a nice natural structural decomposition that corresponds to the syntactic structures of the language. Attribute grammars decorate a tree with

[1] The GEM Tool is available on the world wide web at the location `http://www.icsi.berkeley.edu/~maffy/gem`

attributes. Such attributes may be either inherited or synthesized, which corresponds to a top–down and a bottom–up evaluation, respectively. In general, attribute grammars evaluators may be generated only for attribute grammars with only synthesized attributes. Montages may be interpreted as attribute grammars with only two synthesized attributes, i.e. the Terminal and Initial leaves. In contrast to traditional attribute grammars, these attributes are pointers into the token sequence. We can therefore manipulate attributes in the token sequence resulting in global effects. The restriction of the computation to direct descendent's attributes causes a lack of expressiveness and use of attribute grammars. Moreover, they tend to be very long if applied to realistic languages ([Ode89]).

Static analysis has been already modelled by means of ASMs in previous works. In [BD96] the static analysis and the dynamic semantics of Occam have been specified. The static analysis is performed while building the parse tree in a top–down fashion. The experience with Montages shows that static semantics can only be checked while traversing the tree in a bottom–up fashion taking advantage of the static analysis of the lower levels.

[MJ94] gives a characterization of a number of grammar formalisms, including attribute grammars, by means of ASMs building the parse tree. Our work assumes that the parse tree is given in the initial state.

Using ASMs for dynamic semantics, the work in [PH94] defines a framework comparable to ours. Although it has different aims, namely efficient execution. For the static part, it proposes occurrence algebras which integrate term algebras and context free grammars by providing terms for all nodes of all possible derivation trees. This allows to define all static aspects of the language in a functional algebraic system, which is supported by the MAX tool. back–end of Montages, both for efficient execution of specifications and automated reasoning. The additional mathematical machinery is rather complex and should be hidden from the user. In the existing form the specifications are pretty cryptic.

### Acknowledgments

We gratefully acknowledge Egon Börger, Yuri Gurevich, Jim Huggins, and Eugenio Omodeo for their constructive comments on early drafts of the paper. Thanks goes to David Espinosa, Daniel Schweizer, Chuck Wallace, and Richard Waldinger who helped us with the writing; Richard proposed the name Montages. We are also indebted with the anonymous referees for their insightful comments.

## References

[Anl97a] M. Anlauff. *GEM a Graphical Editor for Montages, User Manual.* ICSI, Berkeley, 1997.

[Anl97b] M. Anlauff. The Semantics of the Object–Oriented Programming Language Sather. Technical report, International Computer Science Institute, Berkeley, 1997. In preparation.

[BCD+87] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. Technical Report 777, INRIA, Sophia Antipolis, 1987.

[BD96] E. Börger and I. Durdanović. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, 39(1):52 – 92, 1996.

[BDR94] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *IFIP 13th World Computer Congress, Volume I: Technology/Foundations*, pages 489 – 508. Elsevier, Amsterdam, 1994.

[BGM95] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA–machines. In *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 1995.

[BR95] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness*, chapter 2, pages 20 – 90. Series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[DCIG93] G. Del Castillo, Durdanović I., and U. Glässer. *An Evolving Algebra Abstract Machine*, volume 1092 of *LNCS*, pages 191 – 214. Springer Verlag, 1993.

[DGKP97] T. Despeyroux, M. Gaieb, P.W. Kutter, and A. Pierantonio. Natural Semantics of Static Aspects of Montages and Generated Tool Support using Centaur. Technical report, INRIA Sophia–Antipolis, 1997. In preparation.

[DiF97] B. DiFranco. Semantica Statica e Dinamica di SQL diretto (ISO 9075) mediante i Montaggi. Master's thesis, Università di L'Aquila, 1997. In preparation (in italian).

[GH93] Y. Gurevich and J.K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274 – 308. Springer Verlag, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Kah87] G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theoretical Aspects of Computer Science, Passau, Germany*, 1987.

[KH97] P.W. Kutter and J.K. Huggins. Abstraction from Evaluation Order and Sequentiality. in preparation, 1997.

[Knu68] D.E. Knuth. Semantics of Context–Free Languages. *Math. Systems Theory*, 2(2):127 – 146, 1968.

[KP97] P.W. Kutter and A. Pierantonio. The formal specification of oberon. *J.UCS*, 3(5), 1997. This volume.

[Kut96] P.W. Kutter. Executable Specification of Oberon Using Natural Semantics. Term Work, ETH Zürich, implementation on the Centaur System [BCD+87], 1996.

[Kut97] P.W. Kutter. Dynamic Semantics of the Programming Language Oberon. TIK-Report 25, ETH Zürich, 1997.

[MJ94] L.S. Moss and D.E. Johnson. Grammar Formalisms Viewed As Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, ETH Zürich, 1989.

[PH94] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 233 – 247. Springer–Verlag, 1994.

[Wal94] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.