Description of Semantics and Code Generation Possibilities for a Multi-Language Interpreter

Guntis Arnicans

Faculty of Physics and Mathematics
University of Latvia
Raina Blvd. 19, Riga LV-1586, Latvia
garnican@lanet.lv

Abstract

In this paper we describe the definition of semantics for a Multi-Language interpreter (MLI), which provides the execution of the given program, receiving and exploiting corresponding language syntax and the desired semantics. We analyze the simplest solution – the MLI receives the language syntax and the semantics descriptions, which have already been compiled to executable objects. Semantics is defined as a composition from several semantic aspects, considering the pragmatics of a language. Semantic aspects are translated to semantic functions by composing descriptions of the aspects. A traversing program's intermediate representation and the calling out of semantic functions similarly to the principle of the Visitor pattern perform the desired semantics. To simplify the semantic descriptions, we use abstract components that are joined by connectors at the meta-level. The implementation of these components and connectors can be very different. Examples of conventional and specific semantics are given for the simple imperative language in this paper.

1. Introduction

The number of new languages that are related to the IT sector has increased rapidly over the last several years (programming languages and data description languages, for example). Problems associated with the implementation and use of these languages has also expanded, of course. Kinnersley [Kin95] has reported that there were 2,000 languages in 1995, which were being put to serious use. Even back then specialists found that the new languages were mostly to be classified as domain-specific languages. Most of them are not easy to implement and maintain [ITSE99, DKV00 (DSL analysis, problems and an annotated bibliography)]. It is also true that we need not just a compiler or an interpreter, but also a number of supportive tools. Questions of programming quality are very important today, and these questions often cannot be answered without specialized and automated ancillary resources.

Computers are being used with increasing dynamism today: systems have been divided up in terms of time and space, the operational environment is heterogeneous, and we have to ensure the implementation of parallel processes while organizing cooperation among components and systems, adapting to changing circumstances without interrupting our work, etc. We are making increasing use of interpreters or of code generation and compilation just in time. The formal resources that are used to describe the semantics of a language, however, cannot fully satisfy our needs in the modern age, and they are starting to lose their positions [Sch97, Lou97, Paa95].

The basic problem that is associated with the formal specifications of programming languages is that these specifications are far too complex. It is not clear how they are administered, we cannot use them to explain all of our practical needs, and in the end we are still faced with a problem – who can prove that these complex specifications are really correct? The literature claims that the best commercial compilers (interpreters or other language-based tools) are written without formalism or are used only in the first phases – scanning and parsing [e.g. Lou97]. Formalisms are elaborated and used mostly for research purposes in educational and scientific institutions at this time.

The development of semantics is gradually moving away from the development of languages and tools. One way to overcome this gap is to take a *tool-oriented approach to semantics*, making the definitions of semantics far more useful and productive in practice and generating as many language-based tools as possible from them [HK00]. We support this approach in principle, but our aim is to propose a different approach toward the definition of semantics, making room for far less formal records.

Those who prepared descriptions of semantics in the past have long since been looking for ways in which semantics can be divided up into reusable components, and it is not yet clear whether the formal or the partly formal methodology is the best in this case. We chose a less formal and more free form of description keeping from the theoretical perspective, and our empirical research showed that rank-and-file developers of tools understand this method far more easily.

2. The concept of a Multi-Language Interpreter

The concept of a *multi-language interpreter* was introduced in [AAB96]. A Multi-Language Interpreter (MLI) is a program which receives source language syntax, source language semantics and a program written in the source language, then performs the operations on the basis of the program and the relevant semantics. Conceptually, we parse an input token stream, build a parse tree and then traverse the tree as needed so as to evaluate the semantic functions that are associated with the parse tree nodes. Once an explicit parse tree is available, we visit the nodes in some order and call out an appropriate function. This approach is similar to the principle *build a tree, save a parse and traverse it* [Cla99] and to a Visitor pattern [GHJV95], except in terms of the methodology which we apply in obtaining semantic functions and organizing physical implementation. The idea of MLI is expressed in Figure 1.

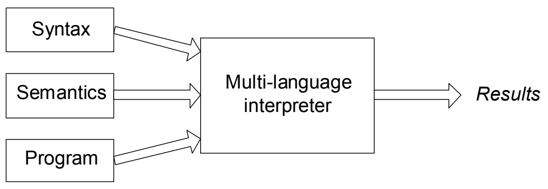


Figure 1. The concept of a Multi-Language Interpreter

The concept of a MLI presupposes that we can prepare several semantics for one syntax, and we can exploit one semantic for various syntaxes. The descriptions of syntaxes and semantics must be translated to the executable form (before or during the running of the MLI). MLI implementation architectures may vary. The one we use receives and exploits syntax and semantic descriptions that have already been compiled as executable objects (Figure 2). Syntax is represented by the *SyntaxObject*, and semantics by the *TraverserObject*, the *SemanticObject*, the *SymbolTable*, and the necessary volume of the *Component* (the components A, B, C in our figure). The *MLI Kernel*, which provides the initial bonding of all syntax and semantics objects, initializes the execution of the program.

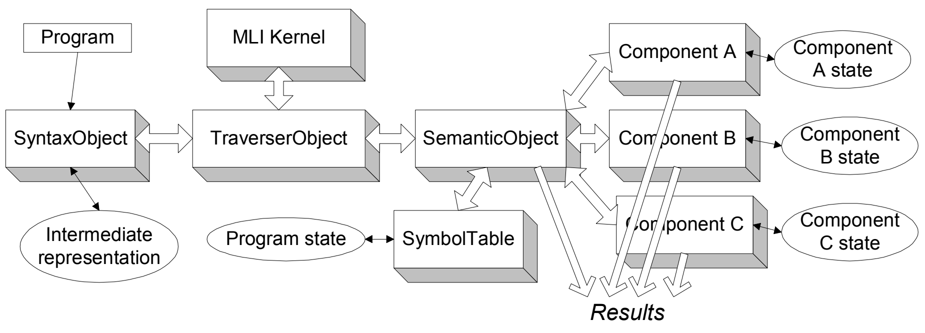


Figure 2. MLI runtime architecture

Each of the components can be implemented in various ways – with a different semantic assignment and physical implementation. Here we have a chance to combine syntaxes and semantics in both ways – in terms of architecture and in terms of implementation. Then, however, we immediately face the question of the compatibility of the syntax and semantics so as to avoid senseless interpretation.

The obtaining of an executable syntax and of semantic objects from their descriptions can be done before or during the actual program execution (analogue to a classical compiler and interpreter). Dynamic code generation is more difficult because all generation phases must be done automatically.

3. Language Specifications for MLI

Programming language is an artificial means to communicate with a computer and to fix the algorithms for problem solving. Like a natural language, a programming language's definition consists of three components or aspects: *syntax*, *semantics* and *pragmatics* [Pag81, SK95]. All of these aspects are significant in dealing with our problems. Usually exploited rarely, pragmatics deals with the practical use of a language, and this is an important element in defining semantics.

We can look at syntax and semantics from two perspectives – the definition or description phase and the runtime phase. Our goal is to achieve runtime components which can freely be exchanged or mixed together in pursuit of the desired collaboration. First we must look at the principles of syntax and semantics descriptions, and then we can view the target code generation steps.

Our basic principle is to divide syntax and semantics into small parts, and later, with a simple method, to combine these parts thus providing a mechanism to tie together the semantic parts and the syntax elements. Our method is close to some of the structuring paradigms of attribute grammars [Paa95]: The definition phase is similar to the relationship *Semantic aspect = Module*, but the runtime phase is similar to *Nonterminal = Procedure*. That means that we basically use the language pragmatics and divide the semantics into semantic aspects.

3.1. Syntax

The formalisms for dealing with the syntax aspect of a programming language are well developed. The theory of scanning, parsing and attribute analysis provides not only the means to perform syntactical analysis, but also a way to generate a whole compiler as well. Such terms, concepts or tools as finite automata, regular expression, context-free grammar, attribute grammar

(AG), Backus-Naur form (BNF), extended BNF (EBNF), Lex (also Flex), Yacc (also Bison), and PCCTS are well known and accepted by the computer science community.

We do not need to reinvent the wheel and it is reasonable to choose the existing formalisms and generators (lexers and parsers). The main task when dealing with syntax description for a given language is code generating which can transform the written program, which uses the syntax, into intermediate representation (IR). Additionally, we need to attach a library with functions, which provide the means to manipulate with the IR and to compile the whole code. The result is the SyntaxObject (Figure 2).

In this paper we concentrate mostly on the class of imperative programming languages, but our method is adaptable for other languages too, such as diagrammatic languages (e.g., Petri nets, E-R diagrams, Statecharts, VPL – visual programming languages, etc.), which exploit other formalisms (e.g., SR Grammars, Reserved Graph Grammar) and processing styles [FNT+97, ZZ97].

3.2. Semantics

The chosen principle for the runtime semantics *parse and traverse* states that the most important things are a traversing strategy and the semantic functions which must be executed when visiting a node (Figure 3). Therefore, the central components of the semantics are TraversalObject and SemanticObject (Figure 2).

The TraversalObject manages the node visiting order, provides semantic functions with information from the IR, and is the main engine of the MLI. The SemanticObject, for its part, contains all of the necessary semantic functions and provides for the execution environment. At the same time, we can also put into the semantic functions certain commands which force the Traverser to search for the needed node and to change the current execution point in the IR (traversing strategy changes and a transition to another node are problems in the Visitor pattern [e.g. Vis01]).

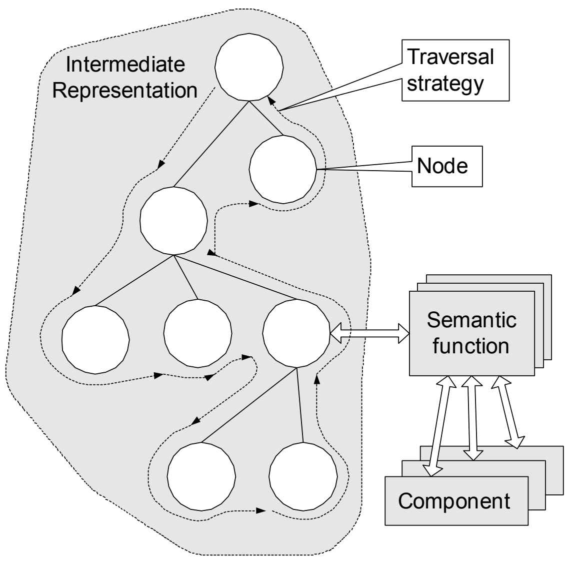


Figure 3. Runtime correspondence between syntax and semantics

Semantic functions have to be as simple and as small as possible. This can be achieved by using a meta-language and by employing high-level expression means, which allow for easy understanding and verification of the description. Following this principle becomes more natural if we use abstract components so that the underlying semantic can be clear without additional explanations (in Figure 3, the abstract components already have a concrete implementation component – A, B and C). This statement may lead to objections from the advocates of formal semantics, because the components are not described with mathematic precision. At the same time, however, formal semantics sometimes use such concepts as *Stack* or *Symbol table*.

Let us introduce a conceptual syntax element, which is a grammar symbol with a name (e.g., a named nonterminal symbol or a named terminal symbol). Considering the various types of syntax elements and the traversing strategy, we separate various visitations and introduce the concept of the traversing aspect. For instance, we can distinguish the arriving into node from the parent node (*PreVisit*) as well the arriving into node from the child node (*PostVisit*). Thus we create the semantic functions and name them not only on the basis of the name of the syntax element but also on the basis of the arriving aspect (traversing aspect) into this element (Figure 3).

Runtime semantics or simply semantics for multi-language interpreters are a set of semantic functions. We represent the runtime semantic in Table 1. There is an executable code (□) or nothing (λ) for the syntax element, according to the traversing aspect. *n* depends on the size of syntax (e.g., the count of all nonterminal and terminal symbols), and *m* depends on the complexity of the traversing strategy (usually 1..3). We notice that the matrix mainly consists of empty functions (λ).

Table 1. The matrix of syntax elements and semantic function correspondence

Syntax Element (SE)	**Traversing Aspect (TA)**				
	T A_1	T A_2	T A_3	.	T A_m
SE_1	□	□	λ	.	λ
SE_2	λ	λ	□	.	λ
SE_3	λ	λ	λ	.	□
…	..	..	..	.	λ
SE_n	□	λ	λ	.	λ

The identification of semantic functions is realized both by the syntax name and by the traversing aspect name. Technical implementation may differ, but it is very advisable that functions identification and calling be performed with constant complexity O(1).

Now we arrive at the most difficult and important problem – how can we obtain semantic functions and ensure correct collaboration between them, and how is it possible to create reusable semantic descriptions? Let us explain our ideas about how to define semantics and how to gain the matrix observed above, i.e., how to generate executable semantics from the semantic description.

4. Semantic Aspects and Abstract Components

4.1. Semantic aspects

In practice, programming languages are frequently presented through the pragmatics of the programming language, i.e., examples are used to show how the language constructs are exploited and what their underlying meaning is. Let us call these language constructs and their meaning like *semantic aspects*.

We have chosen to define the semantic as a set of mutually connected semantic aspects. Here are some examples for typical groups of semantic aspects for imperative programming languages: execution of commands or statements (e.g., basic operations, variable declaring, assigning of a value to the variable, execution of arithmetic expressions), program control flow management (e.g., loop with a counter, conditional loop, conditional branching), dealing with symbols (e.g., variables, constants), environment management (e.g., the scopes of visibility). Here, too, are examples of nontraditional semantic aspects: attractive printing of the program, dynamic accounting of statistics, symbolic execution, specific program instrumentation, etc.

We have chosen an operational approach to describe the semantic aspect – we define the computations, which a computer has to do to perform the semantic action.

4.2. Abstract data types and abstract components

The next significant principle to define the semantic aspect is using *abstract data types* (ADT) as much as possible. ADT is a collection of data type and value definitions and operations on those definitions, which behave as a primitive data type. This software design approach breaks down the problem into components by identifying the public interface and the private implementation.

In our case, typical examples of ADT are *Stack*, *Queue*, *Dictionary*, and *Symbol table* (in compiler construction theory [ASU86, FL88], in formal semantics [SK95]). In this way we hide most of the implementation details and concentrate mainly on the logic of the semantic aspect. Later we can choose the best implementation of ADT for the given task. Seeing that some exploited components can be complicated (*E-mail*, *Graph visualization*, *Distributed communication*, *Transaction manager*, etc.) and have no standards, we use another term – *abstract component*. Sometimes we want to utilize an already existing component, and the term *abstract component* seems more appropriate to us.

It is advisable to describe the semantic aspect through meta-language, even if one does not have a translator for this. Then one can translate or simply rewrite it by hand to the target programming language, select appropriate implementation for the abstract components, and use the needed interface, collaborating protocol and execution environment. For instance, Stack can be implemented in a contiguous memory or in a linked memory, Symbol table – as a list or as a dictionary with the hashing technique. Furthermore, instances of abstract components can be viewed as distributed objects in a heterogeneous computing network.

4.3. Examples of abstract components

Some abstract components and their operations are very popular, e.g., Stack (createStack, push, pop, top, etc.), Queue (createQueue, enqueue, dequeue, first, etc.), while some are guessed, e.g., E-mail (prepare, send, receive, open). Among the many specific components we would like to emphasize one that is useful for most of semantics - Symbol table (SymbolTable in Figure 2) or its analogue to provide the execution environment.

While building prototypes of the MLI, we have created an implementation of Symbol table – MOMS (Memory Object Management System) - that is appropriate for implementing the imperative programming languages. It is possible to define basic and user defined data types, to define base operations and functions, to operate with variables and their values, to manage the scope of visibility of all objects, etc. The most important data types, concepts, and operations of MOMS are listed in the appendix to this paper so as to give the reader a better idea about MOMS.

The second important component is Traverser (TraverserObject in Figure 2). Its main task is realizing the traversing strategy, to change the current execution point and to organize cooperation with the syntax object.

There is a depth-first left-to-right traversing strategy, which is used in the following examples (Table 2). This strategy has three visiting aspects: *Visit* (for tree leaves - terminals), *PreVisit* and *PostVisit* (for the other tree nodes - nonterminals). To define semantic functions for examples, we have used the following operations: *NodeValue()* returns a value for the current terminal or nonterminal symbol (value from the current IR node), and both *goSiblForw(aName)* and *goSiblBackw(aName)* provide for a changing of the current node, searching the node with the name *aName* between siblings going forward or backward.

Table 2. A depth-first left-to-right traversing strategy

```
Traverse(node P)
   if IsLeaf(P)
      Visit(P)
   PreVisit(P)
   for each child Q of P,
   in order, do
      Traverse(Q)
   PostVisit(P)
```

It is possible to describe interfaces for SyntaxObject, TraverserObject and SemanticObject with domain-specific language. Then interfaces for obtaining the IR, manipulating with it and working with the symbol table can be compiled together, and it is possible to engage in high-level optimization and verification [Eng99].

The Traversing strategy can also be described with domain-specific language. This is important if the strategy is not trivial and depends on syntax elements and the program state [OW99 (traversing problems and solutions for Visitor pattern)]. The traversal strategy should be independent from syntax as much as possible and organized (combined) by patterns [Vis01]. In addition to common traversing strategies there are also less traditional ones, e.g., the strategy for reverse execution of the program [BM99]

4.4. Defining the semantic aspect

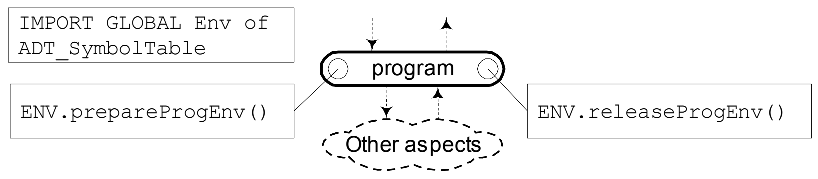
It is more convenient to define the semantic aspect by using diagrams (as in Figure 7). We can write a meta-program or a program in the target language in textual form, too. Diagrams contain syntax elements that are important for the semantic aspect and are visualized with graphic symbols. We can use different graphic notations. If the visiting order of syntax elements is important, then we mark the order with arrows.

Let us call the operations that are performed during the aspect node visiting *semantic action*. Semantic action is similar to semantic function, but it is written at the meta-level and relates only to a given semantic aspect. Semantic action is shown as a box with the meta-code connected to the syntax element and takes into account the traversing aspect.

There are all kinds of abstract data types that are needed for the semantic aspect into the box with the key words *IMPORT GLOBAL*. For better perceptibility of the semantic aspect, it is permissible to use additional graphic symbols that are not needed in real execution. For instance, we use *Other aspects* to signal that we expect there to be a composition with the other semantic aspects.

4.5. Examples of semantic aspects

Let us look at some examples of semantic aspects (Figure 4 - Figure 8) that are applicable for the simple imperative programming language Pam [Pag81]. Terminal symbols are denoted by a rectangle, while nonterminal symbols are indicated by rounded rectangles. The left circle in the nonterminals corresponds to the *PreVisit* semantic action, the right one – to the *PostVisit* semantic action, while for the terminals, the *Visit* semantic action is assigned.

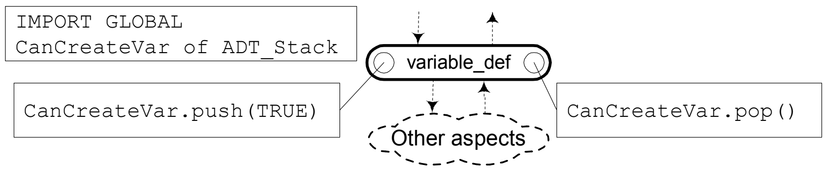


```
IMPORT GLOBAL Env of
ADT_SymbolTable
```

```
ENV.prepareProgEnv()
```

```
ENV.releaseProgEnv()
```

Figure 4. The semantic aspect PROGRAM. It prepares the program environment to manage variables, constants, etc. and operations involving them. The environment is destroyed at the end

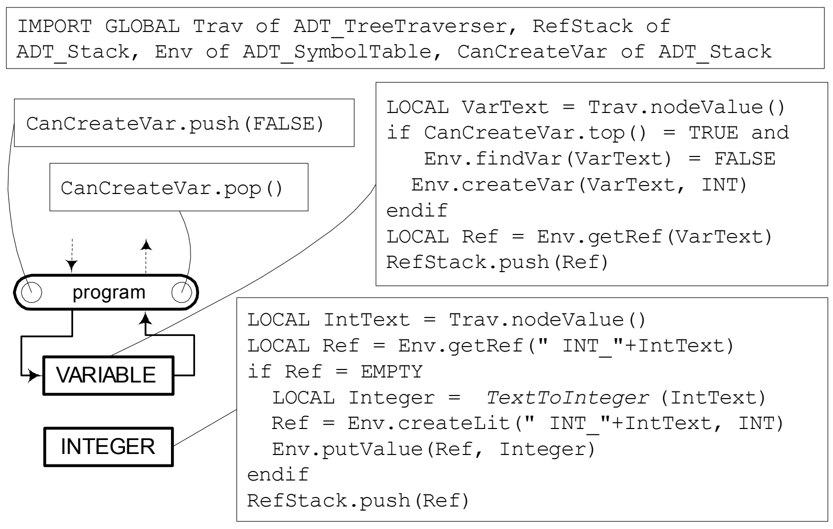


```
IMPORT GLOBAL
CanCreateVar of ADT_Stack
```

```
CanCreateVar.push(TRUE)
```

```
CanCreateVar.pop()
```

Figure 5. The semantic aspect VARIABLE DEFINITION. It allows for variable creation



```
IMPORT GLOBAL Trav of ADT_TreeTraverser, RefStack of
ADT_Stack, Env of ADT_SymbolTable, CanCreateVar of ADT_Stack
```

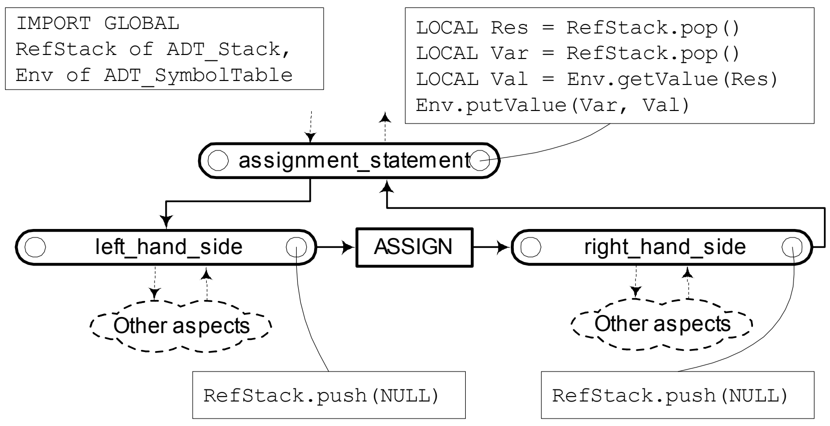
```
CanCreateVar.push(FALSE)
```

```
CanCreateVar.pop()
```

```
LOCAL VarText = Trav.nodeValue()
if CanCreateVar.top() = TRUE and
   Env.findVar(VarText) = FALSE
  Env.createVar(VarText, INT)
endif
LOCAL Ref = Env.getRef(VarText)
RefStack.push(Ref)
```

```
LOCAL IntText = Trav.nodeValue()
LOCAL Ref = Env.getRef(" INT_"+IntText)
if Ref = EMPTY
  LOCAL Integer =  TextToInteger (IntText)
  Ref = Env.createLit(" INT_"+IntText, INT)
  Env.putValue(Ref, Integer)
endif
RefStack.push(Ref)
```

Figure 6. The semantic aspect ELEMENT. It provides for the pushing into the stack all references to each variable encountered while traversing. Variable creation is forbidden by default. The Trav provides for getting the values of the current terminal node in IR.



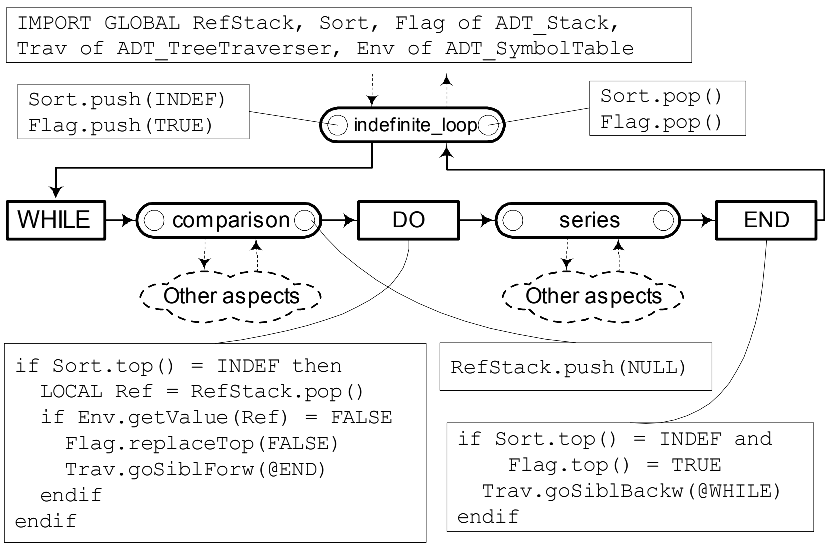
```
IMPORT GLOBAL
RefStack of ADT_Stack,
Env of ADT_SymbolTable
```

```
LOCAL Res = RefStack.pop()
LOCAL Var = RefStack.pop()
LOCAL Val = Env.getValue(Res)
Env.putValue(Var, Val)
```

```
RefStack.push(NULL)
```

```
RefStack.push(NULL)
```

Figure 7. The semantic aspect ASSIGNMENT. It takes reference to the variable and reference to the value from the stack and assigns a value to the variable. Pushing of references is simulated



```
IMPORT GLOBAL RefStack, Sort, Flag of ADT_Stack,
Trav of ADT_TreeTraverser, Env of ADT_SymbolTable
```

```
Sort.push(INDEF)
Flag.push(TRUE)
```

```
Sort.pop()
Flag.pop()
```

```
if Sort.top() = INDEF then
  LOCAL Ref = RefStack.pop()
  if Env.getValue(Ref) = FALSE
    Flag.replaceTop(FALSE)
    Trav.goSiblForw(@END)
  endif
endif
```

```
RefStack.push(NULL)
```

```
if Sort.top() = INDEF and
    Flag.top() = TRUE
  Trav.goSiblBackw(@WHILE)
endif
```

Figure 8. The semantic aspect INDEFINITE LOOP. It “goes through” the series and back to WHILE until the comparison sets a NULL reference or a reference with the value FALSE

5. Meta-semantics

Meta-semantics is a term which relates to a meta-program that describes the counterparts of which semantics consists and the way in which these counterparts are connected together. The conceptual scheme of meta-semantics is shown in Figure 9.

If we look at semantics from the definition side (the logical view) then semantics is formed by traversing strategy and by a set of semantic aspects that consist of semantic actions realized while visiting the appropriate node of the intermediate representation. If we look at semantics from the runtime side (the physical view) then while visiting one node it is possible that several semantic actions have to be realized, and we have to ensure correct collaboration between all involved instances of abstract components into the desired environment. How can we put all of this together correctly?

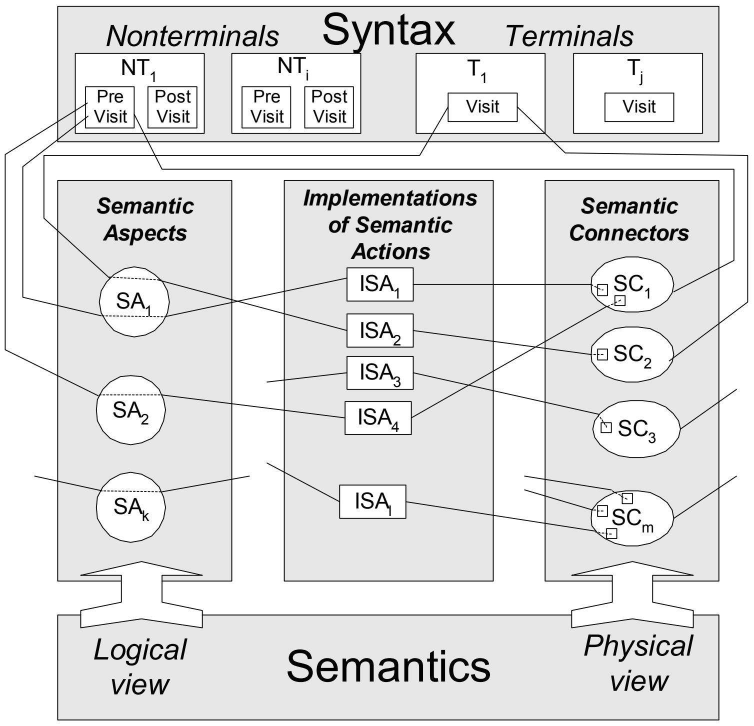


Figure 9. The conceptual scheme of meta-semantics. For instance, SA_1 involves nonterminal NT_1 and terminal T_1, and SC_1 is performed while *previsiting*. SC_1 is a composition of ISA_1 and ISA_4

To solve this problem we introduce the concept of the *semantic connector*. This concept has been adapted from the concept *Grey-box connector*, which solves a similar problem: how to connect the pre-built components in a distributed and heterogeneous environment for collaborating work [AGB00]. The Grey-box connector is a meta-program that introduces a concrete communications connection into a set of components, i.e., it generates the adaptation and communications glue code for a specific connection.

6. From semantic aspects to meta-semantics and executable semantics

The obtaining of semantics for the fixed syntax is achieved in several steps: 1) select predefined semantic aspects or define new ones for the desired semantics, 2) rename syntax elements and traversing aspect in the selected semantic aspects with names from fixed syntax and traversing strategy, 3) rename instances of abstract components to organize collaboration between semantic aspects, 4) make composition from semantic aspects, 5) specify the runtime environment and translate the meta-code to the code of the target programming language, and 6) compile the semantics.

- Selection of semantic aspects (step 1)

If we have a library with previously created semantic aspects, then we can search for appropriate ones, i.e. reuse some parts of the semantics. The traversing strategy also has to be considered.

- Syntax element and traversing aspect renaming in semantic aspects (step 2)

Actually this is semantic action mapping. Matching to the fixed syntax elements is achieved by mapping syntax elements of semantic aspects to fixed syntax elements (*rename with* – simple mapping and *duplicate to* – mapping of a semantic aspect syntax element to several fixed syntax elements):

rename <name> <traversing aspect> *with* <target name> <target traversing aspect>
duplicate <name> <traversing aspect > *to* <target name> <target visiting aspect>
[,<target name> <target visiting aspect> …]

For instance, *rename* left_hand_side PostVisit *with* VARIABLE Visit ;

duplicate COUNTABLENODE Visit *to* VARIABLE Visit, assignment_statement PostVisit

- Renaming of instances of abstract components (step 3)

Matching of components is necessary because there is no direct data exchange between semantic functions, and we need collaborative work. The program state is fixed by using the runtime states of components. At first we decide what instances they have in common and what names they have to get, and then we rename instances in the semantic aspects:

replace <name> *with* <target name>

For instance, *replace* RefStack *with* DataStack ; *replace* Sort *with* LoopSortStack

- Composition of semantic aspects (step 4)

The goal of a semantic aspect composition is to bring together several semantic aspects into one more complicated aspect that nearly describes entire semantics. While composing, we stick together the meta-code of semantic actions that have the same name. The sticking principles can vary, for instance, *sequential* (one code is appended to the other one), *parallel* (codes can be executed simultaneously or sequentially in any order), *free* (the user can modify the code union as he likes). To achieve better results, we ignore some semantic actions or apply the sticking principle to the semantic aspects in the reverse order. At this time we have to be aware of conflicts between local variable names.

compose aspect <<new SA>> (<refined SA>)
    [[*append* | *parallel*, *free*] (<refined SA>) …] , where

<refined SA> = <<old SA>> [*ignore* <name> <trav aspect>
    [,<name> <trav aspect>]] | [*reverse*]

The result is meta-semantics. An example of a meta-code fragment for meta-semantics is given in Table 3.

- Meta-code translating (step 5)

Meta-code is translated to the target programming language, taking into account the target language (e.g. C++), the implementation of abstract components (e.g. Stack in linked memory), the operating system (e.g. Unix), the communications between components (e.g. CORBA), MLI components type (e.g. DLL), etc. The translation may be done by hand or automatically (desirable in common cases).

- Obtaining semantic objects (step 6)

By compiling the code we get executable objects that provide semantic performance, i.e. they contain the semantic functions that are called while traversing the program intermediate representation. The instances of the concrete implementation of abstract components are created, or the existing ones are dynamically linked via the selected communications protocols.

Table 3. An example of meta-semantics

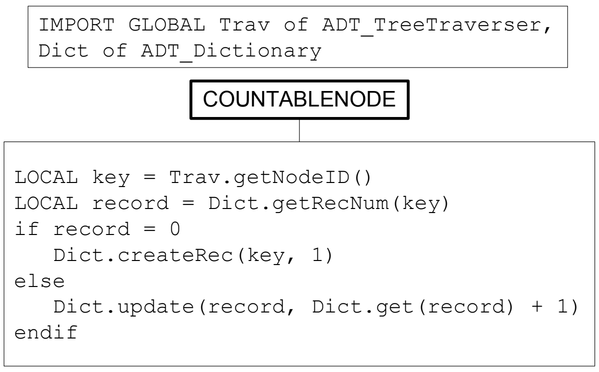
```
compatible with
  ir_type ParseTree
  traverser_type ParseTreeTraverser

syntax elements (program, expression, VARIABLE, ...)
semantic actions (<PROGRAM> program PreVisit
{ENV.prepareProgEnv()},
                  <PROGRAM> program PostVisit {...}, ...)

global Trav of ADT_TreeTraverser
global Env of ADT_SymbolTable
create DataStack, OperatorStack, CanCreateVar, LoopSortStack,
    LoopCounterStack, LoopFlagStack, IfFlagStack of ADT_Stack
create InputFile, OutputFile of ADT_FILE

compose aspect <A1>     // composes semantic aspects from aspects given above
  (<PROGRAM>)           // semantic aspects PROGRAM remains the same
append (<ELEMENT>
  replace RefStack with DataStack // replaces stack for collaborating work
  rename INTEGER Visit with CONSTANT Visit)    // renames nonterminal according
to PAM syntax
append (<ASSIGNMENT>
  replace RefStack with DataStack
  rename left_hand_side PostVisit with VARIABLE Visit,
         right_hand_side PostVisit with expression PostVisit
  ignore left_hand_side PostVisit)       // ignore pushing of NULL reference
append (<INDEFINITE LOOP>
  replace RefStack with DataStack,
    Sort with LoopSortStack, Flag with LoopFlagStack)
append (<VARIABLE DEFINITION>
  rename variable_def PreVisit with assign_statement PreVisit,
    variable_def PostVisit with ASSIGN Visit)
end compose aspect

compose aspect <A2>
(<A1>
  ignore expression PostVisit, comparision PostVisit)
  append ( ...
  /* Others aspect are appended such as <TYPE AND OPERATOR>,
<INPUT>, <OUTPUT>, <BASE BYNARY OPERATION>, <DEFINITE LOOP>,
<CONDITIONAL STATEMENT> */
  ...)
end compose aspect
```

7. Examples of alternative semantic aspects

In this section we provide a short insight on how we can build nontraditional semantics. By adding new features to existing semantics we can create a specific tool that works with a given programming language. We would like briefly to survey two examples: 1) statistics accounting of program point visiting, and 2) storing of symbolic values for variables. Both aspects are added to the conventional semantics, and this is program instrumentation if we speak in terms of software testing.

7.1. Accounting of program point visiting

Our goal is to account for any visiting of a desirable program point. This means that we need to set counters at these points. At first we write the semantic aspect NODE COUNTER (Figure 10). We use the abstract component *Dictionary* where we can store, read and update records in form <*key, value*>.



```
IMPORT GLOBAL Trav of ADT_TreeTraverser,
Dict of ADT_Dictionary
```

```
LOCAL key = Trav.getNodeID()
LOCAL record = Dict.getRecNum(key)
if record = 0
   Dict.createRec(key, 1)
else
   Dict.update(record, Dict.get(record) + 1)
endif
```

Figure 10. The semantic aspect NODE COUNTER

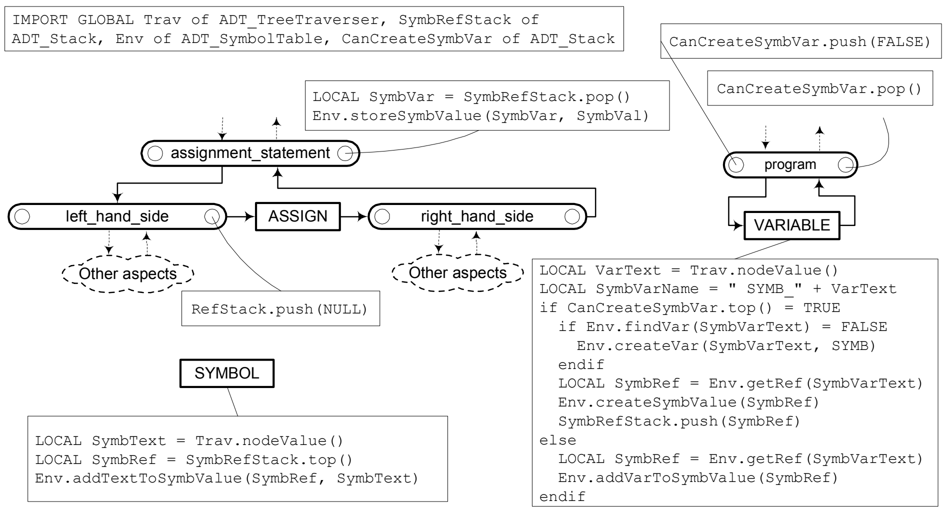
Defining meta-semantics, we add this semantic aspect to the others. For instance, we define accounting for the use of any variable and assignment operation:

dublicate countable_node Visit *to* VARIABLE Visit, assignment_statement PostVisit

Another semantic aspect can be built which accounts for every concrete variable using statistics into an additional dictionary (the variable name serves as the key). We can improve this aspect further by accounting for an aspect of variable use - defined, modified, referenced, released, etc. In the program analysis and instrumentation area, our approach is similar to the Wyong system (based on the Eli compiler generation system and the ATOM program instrumentation system), because specific operations are attached to syntax elements, and in this way we obtain a specific tool with additional semantics [Slo97].

7.2. Storing of symbolic values for variables

The second example provides for the fixing of symbolic values for variables (Figure 11). To do this task in an effective way, we have additional operations in our MOMS (symbol table). The operation *createSymbValue* creates an entry for symbolic value, and with the operations *addTextToSymbValue* and *addVarToSymbValue*, we form the value while traversing all nodes in the desired subtree (we store all needed program symbols and symbolic values of variables). At the end we store accumulated value with the operation *storeSymbValue*.



```
IMPORT GLOBAL Trav of ADT_TreeTraverser, SymbRefStack of
ADT_Stack, Env of ADT_SymbolTable, CanCreateSymbVar of ADT_Stack
```

```
CanCreateSymbVar.push(FALSE)
```

```
CanCreateSymbVar.pop()
```

```
LOCAL SymbVar = SymbRefStack.pop()
Env.storeSymbValue(SymbVar, SymbVal)
```

```
RefStack.push(NULL)
```

```
LOCAL SymbText = Trav.nodeValue()
LOCAL SymbRef = SymbRefStack.top()
Env.addTextToSymbValue(SymbRef, SymbText)
```

```
LOCAL VarText = Trav.nodeValue()
LOCAL SymbVarName = " SYMB_" + VarText
if CanCreateSymbVar.top() = TRUE
  if Env.findVar(SymbVarText) = FALSE
    Env.createVar(SymbVarText, SYMB)
  endif
  LOCAL SymbRef = Env.getRef(SymbVarText)
  Env.createSymbValue(SymbRef)
  SymbRefStack.push(SymbRef)
else
  LOCAL SymbRef = Env.getRef(SymbVarText)
  Env.addVarToSymbValue(SymbRef)
endif
```

Figure 11. The semantic aspect SYMBOLIC VALUES

8. Conclusions

This method was developed with the goal of reducing the gap between practitioners (tool developers) and theoreticians (developers of formal specifications for semantics). Our experience shows that a significant achievement is attained if abstract components or abstract data types realize the greater part of semantics, because that way it is easier to perceive the full implementation of semantics.

The second acquisition of our method involves significant disjoining of syntax and semantics from each other. It allows us to combine various syntaxes and semantics and to find out the most desirable semantics for the given syntax. So, if we have written syntax for a new language, we can match several semantics to it in a comparatively short time. As a result, we can develop a wide spectrum of tools in support of our new language.

Our approach allows us to change semantics dynamically while the interpreter is running, i.e. replace semantics or execute various ones simultaneously. It is possible to reduce a derived parse tree (by deleting nodes with empty connectors) or to optimize it (tree restructuring statically and dynamically, considering performance statistics).

At this moment the environment for tool construction or semantics generation is not completely developed. Our experience shows that tools can be developed without significant investments, for example, by using Lex/YACC as a generator to create a syntactic object, which produces program intermediate representation. It is not too difficult to develop a Traverser and a simple SymbolTable. And as the last job, we have to work up semantic aspects on the basis of our method and compose them, thus obtaining connectors, which can be written in some common programming language (skipping meta-language use and its translation). The use of abstract components depends on target semantics.

We believe that the development of the serious tools demands a more universal implementation of the Symbol table. Our Symbol table implementation - MOMS - is not applicable only for imperative language implementations. It was also the basic object-oriented database for the commercial application Mosaik (Sietec consulting GmbH Co. OHG, graphical CASE tool for business modelling).

The weakness of our method lies in the semantic aspects composition stage. At this moment we have not analyzed all risks in terms of obtaining senseless or erroneous semantics. The problems are not trivial, and they are similar to problems in the proper collaboration of objects or components in object-oriented programming, too. [e.g. ML98]. Most name conflicts can be precluded automatically, but it is considerably harder to organize collaboration among the common components in semantic aspects (it is easier if the semantic aspects are mutually independent).

Another problem is that the language grammar is frequently not context free (this is true of our example above, too). In this case we have to introduce additional flags to memorize the context of syntax elements. It is advisable to rewrite the syntax and to use context-free grammars.

9. References

[AAB96] V. Arnicane, G. Arnicans, and J. Bicevskis. Multilanguage interpreter. In H.-M. Haav and B. Thalheim, editors, *Proceedings of the Second International Baltic Workshop on Databases and Information Systems (DB&IS '96)*, Volume 2: Technology Track, pages 173-174. Tampere University of Technology Press, 1996.

[AGB00] U. Aßmann, T. Genßler, and H. Bär. Meta-programming Grey-box Connectors. *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pp.300-311, 2000.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Technigues, and Tools*. Addison-Wesley, 1986.

[BM99] B. Biswas and R. Mall. Reverse Execution of Programs. *ACM SIGPLAN Notices*, 34(4):61-69, April 2000.

[Cla99] C. Clark. Build a Tree – Save a Parse. *ACM SIGPLAN Notices*, 34(4):19-24, April 2000.

[DKV00] A. Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26-36, June 2000.

[Eng99] Dawson R. Engler. Interface Compilation: Steps toward Compiling Program Interfaces as Languages. In DSL-99 [ITSE99], pp.387-400.

[FL88] Charles N. Fisher, and Richard J. LeBlanc, Jr. *Crafting A Compiler*. Benjamin-Cummings, 1988.

[FNT+97]F. Ferrucci, F. Napolitano, G. Tortora, M. Tucci, and G. Vitiello. An Interpreter for Diagrammatic Languages Based on SR Grammars. *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, pages 292-299, 1997.

[GHJV95]E. Gamma, R. Helm, R. Johnson, and J. Vlisides. *Design Patterns: Elements of Reusable Software*, pages 331-334. Addison-Wesley, 1995.

[HK00] J. Heering and P. Klint. Semantics of Programming Languages: A Tool-Oriented Approach. *ACM SIGPLAN Notices*, 35(3):39-48, March 2000.

[ITSE99] Special issue on domain-specific languages. *IEEE Transactions on Software Engineering*, 25(3), May/June1999.

[Kin95] W.Kinnersley, ed., The Language List. 1995. http://wuarchive.wustl.edu/doc/misc/lang-list.txt

[Lou97] Kenneth C. Louden. Compilers and Interpreters. In Tucker [Tuc97], pp.2120-2147.

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. SIGPLAN Notices, 33(10):97-116, 1998. *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '98)*.

[OW99] J. Ovlinger and M. Wand. A Language for Specifying Recursive Traversals of Object Structures. SIGPLAN Notices, 34(10):70-81, 1999. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)*.

[Paa95] J. Paakki. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196-255, June 1995.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice-Hall, 1981.

[Sch97] David A. Schmidt. Programming Language Semantics. In Tucker [Tuc97], pp.2237-2254.

[SK95] K. Slonneger and B. L. Kurtz. *Formal Syntax and semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesly, 1995.

[Slo97] A. M. Sloane. Generating Dynamic Program Analysis Tools. *Proceedings of the Autralian Software Endineering Conference (ASWEC'97)*, pp.166-173, 1997.

[Tuc97] Allen B. Tucker, editor. *The computer science and engineering handbook*. CRC Press, 1997.

[Vis01] J. Visser. Visitor Combination and Traversal Control. SIGPLAN Notices, 36(11):270-282, 2001. *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01)*.

[ZZ97] D.-Q.Zhang and K.Zhang. Reserved Graph Grammar: A Specification Tool For Diagrammatic VPLs. *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, pages 292-299, 1997.

10. Appendix

Table 4. MOMS types

Type	Description
Constructor	Handle to an object type description
Value	Handle to a byte stream that contains an object value
Reference	Handle to a memory object
MemoryMap	Main central object that organizes other objects (other MemoryMap also)
IdentifDict	Dictionary of all identifiers (variables, constants, etc.)
TypesDict	Dictionary of all types (basic types and user defined)
FunctDict	Dictionary of all functions (basic operators, basic functions, user defined functions)
NamesTable	Compact storing of all strings
ConstrTable	Compact storing of all constructor descriptions
ValuesTable	Storing and managing of all object values
MemoryBlock	Organizing scope visibility in all dictionaries and providing memory management
…Others	Stack, queue, collection, etc.

Table 5. Defining of user defined data types

Operation	Description
ConstrPtr **createConstrArray**(Uint minIndex, Uint maxIndex, ConstrPtr ptrToElemConstr)	Defines an array type with the given dimensions and element types. Here and in other functions we can use any previously defined (or partly defined) data type.
ConstrPtr **createConstrFunct**(ConstrPtr ptrToReturnConstr, ConstrPtr ptrToParamConstr)	Defines a function type with the given parameters and return type.
ConstrPtr **createConstrName**(char* aName, ConstrPtr trToSubConstr)	Assigns a user-defined name for the given type.
ConstrPtr **createConstrPointer**(ConstrPtr ptrToSubConstr)	Defines a pointer type to the given type.
ConstrPtr **createConstrProduct**(ConstrPtr ptrToSubConstr1, ConstrPtr ptrToSubConstr2)	Creates a production of two types (establishes some relation between them). It is useful to construct a serious data structure.
ConstrPtr **createConstrRecord**(ConstrPtr ptrToSubConstr)	Defines a record data type (a set of pairs {name, type}).
… Other constructors	For instance, base data type constructors
constrArraySetMinIndex(ConstrPtr ptrToConstr, Uint minIndex)	Modifies the type description (attributes).
Uint **constrArrayGetMinIndex**(ConstrPtr ptrToConstr)	Provides details about data type attributes.
… Others	

Table 6. System initialization and global operations

Operation	Description
MemoryMap **initialize**(Uint memoryCount)	Creates MOMS with internal parallel but related memories (MOMS)
Uint **switchMemoryTo**(Uint memNum)	We can exploit only the specific internal memory
Uint **getCurrentMemory**(void)	Returns the number of the actual memory
defineCountOfBaseTypes(Uint countOfBaseTypes)	Defines a count of the basic types of MOMS
defineBaseType(char* typeName, Uint type, Uint typeSize)	Defines base types. This interface is in C and depends on previously defined types. Some examples: defineBaseType("long_", LONG_, sizeof(long_)); defineBaseType("boolean_", BOOLEAN_, sizeof(boolean_)); defineBaseType("date_", DATE_, sizeof(date_))
defineBaseFunction(char* langFunctName, char* internalName, Uint returnType, int paramCount, ...)	Defines the base operations and functions. This interface is in C and depends on previously defined types. Some examples: defineBaseFunction("+", "PLUS", LONG_, 2, LONG_, LONG_); defineBaseFunction("day", "day", LONG_, 1, DATE_)
prepareProgramEnv(Uchar scope)	Prepares a new MemoryBlock, defines the scope (visibility) of previously defined variables, types, functions
releaseProgramEnv(void)	Releases a current MemoryBlock and all related memory in other objects (dictionaries, tables) and restores a previously defined MemoryBlock
defineAutomaticMemSwitching(Uint firstMemNum, Uint lastMemNum)	Provides for automatic switching in various functions. For instance, we look up the variable in a local memory and then in a global memory (if the variable is not founded yet).
… Others	

Table 7. Operations with variables and similar objects

Operation	Description
createVar(char* aName, ConstrPtr ptrToConstr)	Creates a variable with the given name and type.
createVar(char* aName, char* typeName)	Creates a variable with the given name and type name.
createLiteral(char* aName, ConstrPtr ptrToConstr)	Creates a literal (constant) with the given name and type.
Ref **createRef**(ConstrPtr ptrToConstr)	Creates an object without a name with the given type, for instance, internal loop counter, return value of function.
ConstrPtr **getConstrRef**(char* typeName)	Returns pointer to type with the given type name.
createSynonym(char* aName, Ref aRef)	Creates another reference by name to the existing object.
… Other	

Table 8. Operations with value

Operation	Description
putValue(Ref aRef, char* aValue)	Sets a new value for the object.
char* **getValue**(Ref aRef)	Returns a value for the object.
char* **createDynamicValue**(ConstrPtr ptrToConstr)	Provides dynamic memory allocation for the object given by type.
deleteDynamicValue(char* aValue)	Releases dynamically allocated memory.
setValueProtectionOn(char* aName)	Protects a value of the given object against modification, for instance, protects constants.
setValueProtectionOff(char* aName)	Takes off a value protection.
gotoArrayElementConstr(Ref& aRef, Sint index)	Sets a virtual mark to element constructor and to a given array element value. aRef is modified, it refers to the array element.
gotoNameConstr(Ref& aRef)	Moves the virtual mark to the name constructor.
gotoPointerConstr(Ref& aRef)	Moves the virtual mark to the pointer subconstructor and to the start of value.
gotoProductionLeftConstr(Ref& aRef)	Moves the virtual mark to the left subconstructor and to the start of the corresponding value.
gotoProductionRightConstr(Ref& aRef)	Moves the virtual mark to the right subconstructor and to the start of the corresponding value.
gotoRecordConstr(Ref& aRef)	Moves the virtual mark to the start of record.
gotoNameInList(Ref& aRef, char* aName)	Moves to the appropriate type and value (list of named types linked by productions) E.g., search a field in the user-defined structure.
… Other	