Tom Language Reference

Pierre-Etienne Moreau
(with Julien Guyon, Anne-Claire Lonchamp, and Antoine Reilles)

July 29, 2005

This manual contains information for Tom version 2.2.
This manual also exists in Postscript or pdf.

Tom is a language extension which adds new matching primitives to languages like C, Java, and Caml. Although rich and complex, Tom is not a stand-alone language: like a preprocessor, it strongly relies on the underlying language (C, Java, or Caml), called *host-language* in the following. To this language, Tom adds several constructs.

The main construct, %match, is similar to the match primitive found in functional languages: given an object (called subject) and a list of patterns-actions, the match primitive selects the first pattern that matches the subject and performs the associated action. The subject against which we match can be any object, but in practice, this object is usually a tree-based data-structure, also called *term* in the algebraic programming community.

The match construct may be seen as an extension of the classical switch/case construct. The main difference is that the discrimination occurs on a term and not on atomic values like characters or integers: the patterns are used to discriminate and retrieve informations from an algebraic data structure. Therefore, Tom is a good language for programming by pattern matching, and it is particularly well-suited for programming various transformations on trees/terms or XML data-structures.

Information on Tom is available at the Tom web page tom.loria.fr.

1 The core language

Writing a good documentation is a difficult task. To help new users, as well as confirmed Tom developers, we have split the documentation into four complementary documents:

- the *reference manual* is a document which describes the language constructs, gives their precise syntax and informal semantics. To support the intuition, some examples may be given, but it is by no means a tutorial introduction to the language.
- the *user guide* describes how to use the Tom system in a production environment. This document explains both, how to install the software, and how to use it. In particular, the user guide describes the runtime library and external tools, like *vas*, which facilitates the use of Tom.
- the *tutorial* introduces, using examples, many concepts related to Tom, from simple to complex ones. The goal is to illustrate concepts and ideas introduced by the reference and the user manual.
- the *cookbook* is a collection of recipes that we use, and can be adapted to solve frequently encountered problems.

This document is intended as a reference manual for the Tom language, and thus precisely describes the constructs and their semantic.

2 Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (‘like this’). Non-terminal symbols are set in italic font (*like that*). Square brackets [...] denote optional components. Parentheses with a trailing star sign (...)* denotes zero, one or several repetitions of the enclosed components. Parentheses with a trailing plus sign (...)+ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

2.1 Lexical conventions

Identifier ::= *Letter* (*Letter* | *Digit* | ‘_’ | ‘-’)*
Integer ::= (*Digit*)+
Double ::= (*Digit*)+ [‘.’] (*Digit*)* | ‘.’ (*Digit*)+
String ::= ‘"’ (*Letter* | (‘\’ (‘n’ | ‘t’ | ‘b’ | ‘r’ | ‘f’ | ‘\’ | ‘’’ | ‘"’)))* ‘"’
Letter ::= ‘A’ ... ‘Z’ | ‘a’ ... ‘z’
Digit ::= ‘0’ ... ‘9’
Char ::= ‘’’ (*Letter* |*Digit*) ‘’’

2.2 Names

SubjectName ::= *Identifier*
Type ::= *Identifier*
SlotName ::= *Identifier*
HeadSymbol ::= *Identifier*
| *Integer*
| *Double*
| *String*
| *Char*
VariableName ::= *Identifier*
AnnotedName ::= *Identifier*
LabelName ::= *Identifier*
FileName ::= *Identifier*
AttributeName ::= *Identifier*
XMLName ::= *Identifier*
Name ::= *Identifier*

3 Tom constructs

A Tom program is a host language program (namely C, Java, or Caml) extended by several new constructs such as %match, %rule, %include, %vas, or backquote. Tom is a multi-languages compiler, so, its syntax depends on the host language syntax. But for simplicity, we only present the syntax of its constructs and explain how they can be integrated into the host language.

Using Java as *host-language*, the following Tom program is correct:

```
public class HelloWorld {
  %include { string.tom }

  public String getWord(String t) {
    %match(String t) {
      "World"  -> { return "World";}
      _        -> { return "Unknown"; }
    }
  }

  public final static void main(String[] args) {
    HelloWorld o = new HelloWorld();
    System.out.println("Hello " + o.getWord("World"));
```

```
  }
}
```

3.1 Tom program

Basically, a Tom program is list of blocks, where each block is either a Tom construct, or a sequence of characters. The idea is that that after transformation, the sequence of characters merged with the compiled Tom constructs should be a valid host language program. In the previous example, %include and %match constructs are replaced by function definitions and Java instructions, making the resulting program a correct Java program.

Syntax of a Tom program:

Tom ::= *BlockList*
BlockList ::=
(*MatchConstruct*
| *RuleConstruct*
| *BackQuoteTerm*
| *IncludeConstruct*
| *VasConstruct*
| *TypeTerm*
| *Operator*
| *OperatorList*
| *OperatorArray*
| '{' *BlockList* '}'
)*

- *MatchConstruct* is translated into a list of instructions. This construct may appear anywhere a list of instructions is valid in the host language.
- *RuleConstruct* is translated into a function definition. This construct may appear anywhere a function declaration is valid in the host language.
- *BackQuoteTerm* is translated into a function call.
- *IncludeConstruct* is replaced by the content of the file referenced by the construct. If the file contains some Tom constructs, they are expanded.
- *VasConstruct* allows to define a VAS grammar. This construct is replaced by the content of the generated mapping. See the *user guide* for more details.
- *TypeTerm*, as well as *Operator*, *OperatorList*, and *OperatorArray* are replaced by some functions definitions.

3.2 Match construct

The %match construct (*MatchConstruct*) is one of the main contributions of Tom. This construct can be seen as an extension of the *SwitchCase* construct in C or Java, except that patterns are no longer restricted to constants (chars or integers). Given an object (the subject) and a list of patterns, our goal is to find the first pattern that *match* the subjects (i.e. that have a *compatible shape*). More formally, a pattern is a term built over variables and constructors. The latter ones describe the *shape* of the pattern, whereas the former ones are *holes* that can be instantiated to capture a value. When we consider the term $f(a, g(b))$, this has to be viewed as a tree based data-structure where a is the first child of the root, labeled by f. Similarly, b is the unique child of g, which is the second child of the root. We say that the pattern $f(x, y)$ matched this term (called subject), because we can give values to x and y such that the pattern and the subject become equal: we just have to assign a to x and $g(b)$ to y. Finding this assignment is called matching and instantiating. This is exactly what Tom is supposed to do. A pattern may of course contains subterms. Therefore, $f(x, g(b))$ or $f(a, g(y))$ are valid patterns which match against the subject.

Assuming that s is a Java variable of sort T, which references a term (the tree based object $f(a, g(b))$ for example), the following Tom construct is valid:

```
%match(T s) {
  f(a,g(y)) -> { /* code that uses y */ }
  f(x,g(b)) -> { /* code that uses x */ }
  f(x,y)    -> { /* code that uses x and y */ }
}
```

A *MatchConstruct* is composed of two parts:

- a list of host language variables (called *subjects*). These variables should reference the objects to be matched.
- a list of *PatternAction*: this is a list of pairs (pattern,action), where an action is a set of host language instructions which is executed each time a pattern matches the subjects.

The construct is defined as follow:

MatchConstruct	::=	'%match' '(' *MatchArguments* ')' '{' (*PatternAction*)* '}'
MatchArguments	::=	*Type SubjectName* (',' *Type SubjectName*)*
PatternAction	::=	[*LabelName*':'] *TermList* ('|' *TermList*)* '->' '{' *BlockList* '}'
TermList	::=	*Term* (',' *Term*)*

For expository reasons, we consider that a %match construct is evaluated in the following way:

- given a list of host language variables (they corresponds to objects only composed of constructors, called ground terms), the execution control is transferred to the first *PatternAction* whose patterns match the list of ground terms.
- given such a *PatternAction*, the list of variables is instantiated and the associated semantic action is executed. The instantiated variable are bound in the underlying host-language, and thus can be used in the action part.
- if the execution control is transferred outside the %match instruction (by a goto, break or return for example), the matching process is finished. Otherwise, the execution control is transferred to the next *PatternAction* whose patterns match the list of ground terms.
- when there is no more *PatternAction* whose patterns match the list of subjects, the %match instruction is finished, and the execution control is transferred to the next instruction.

The semantics of a match constructs may remind the switch/case construct, however, a big difference exists. In Tom, a theory like associativity may be attached to a constructor. Thus, given a subject and a pattern, there may exist several ways to match the pattern against the subject. Informally, when considering the subject $conc(a, g(b), c, g(d))$, and the pattern $conc(\_^*, g(x), \_^*)$, there are two possible match for this problem: either $x = b$, either $x = d$. note that $\_^*$ is a special *hole* which can capture any sublist of $conc(\ldots)$.

When taking this new possibility into account, the evaluation of a %match construct is a bit more complex:

- given a *PatternAction* whose patterns match the list of ground terms, the list of variables is instantiated and the associated semantic action is executed.
- if the execution control is not transferred outside the %match instruction, in addition to the previous explanations, if the considered matching theory may return several matches, for each match, the free variables are instantiated and the associated semantic action is executed. This means that a same action may be executed several times, but in a different context: i.e. the variable have different instantiations.
- when all matches have been computed (there is at most one match in the syntactic theory), the execution control is transferred to the next *PatternAction* whose patterns match the list of ground terms.

- as before, when there is no more *PatternAction* whose patterns match the list of subject, the %match instruction is finished, and the execution control is transferred to the next instruction.

As mentioned in the BNF-syntax, a *label* may be attached to a pattern. In that case, in C and Java, it becomes possible to exit from the current *PatternAction* (using a goto or a break), without exiting from the whole %match construct. The control is transferred to the next pattern which matches the subjects. This feature is useful to exit from a complex associative-matching pattern which, for any reason, is no longer interesting.
Note: the behavior is not determined if a semantic action modifies a host language variable which is an argument of a %match instruction under evaluation.
Note: the ''|'' symbol can be used to introduce disjunction of pattern. This is a shortcut which avoid duplicating a *PatternAction*. However, this construct is **deprecated**. We recommend to use the *disjunction of symbols* construct, presented in the next section.

3.3 Tom pattern

As we can imagine, the behavior of a %match construct strongly depends on the patterns which are involved. The formalism which defines the syntax of a pattern is also an essential component of Tom. Unfortunately, its formal definition is complex, simply because there exist several ways to define patterns which have equivalent behaviors. On the other hand, the different shortcuts may help the programmer to simplify the definitions of patterns.

Informally, a pattern is a term that can be either a variable or an anonymous variable (x or _ for example). A pattern can also be composed of constructors (a, f(a), g(a,x), or h(a,_,x) for example). When a pattern matches a subject, it may be useful to keep a reference to a matched subterm. The annotation mechanism (z@g(y) for example) can be used for this purpose. Thus, considering the pattern $f(x, z@g(y))$ and the subject $f(a, g(h(b)))$, y is instantiated by $h(b)$, and z is instantiated by $g(h(b))$. This can be useful in C, to free the memory for example.

When identical actions have to be performed for a set of patterns which share a common structure, the *disjunction of symbols* may be used: pattern $(f \quad | \ g)(a)$ is equivalent to the set $\{f(a), g(a)\}$. The disjunction of symbols may also be used in subterm, like in $h((f \quad | \ g)(x))$.

More formally, a Tom term has the following syntax:

Term	::=	[*AnnotedName* '@'] *PlainTerm*
PlainTerm	::=	*VariableName* ['*']
	|	*HeadSymbolList* [*ExplicitTermList* | *ImplicitPairList*]
	|	*ExplicitTermList*
	|	'_'
	|	'_*'
	|	*XMLTerm*
HeadSymbolList	::=	*HeadSymbol*
	|	'(' *HeadSymbol* ('|' *HeadSymbol*)+ ')'
ExplicitTermList	::=	'(' *Term* (',' *Term*)* ')'
ImplicitPairList	::=	'[' *PairTerm* (',' *PairTerm*)* ']'
PairTerm	::=	'[' *SlotName* '=' *Term* ']'

A pattern is a term which could contain variables. When matching a pattern against a subject (a ground term), these variables are instantiated by the matching procedure (generated by Tom). In Tom, the variables do not have to be declared: their type is inferred automatically, depending on the context in which they appear.

As described previously, Tom offers several mechanisms to simplify the definition of a pattern:

- standard notation: a pattern can be defined using a classical prefix term notation. To make a distinction between variables and constants, it is recommended to explicitly write the empty list of arguments: for example, x(), denotes the constant x. In this case, the corresponding Tom operator (%op x) should have been declared. For simplicity, it is also possible to use the notation x, but note that the status of x depends on the existence of a Tom operator or not: x is a constant if %op x is defined, otherwise it is a variable.

- unnamed variable: the _ notation denotes an anonymous variable. It can be used everywhere a variable name can be used. It is useful when the instance of the variable does not need to be used. Similarly, the _* notation can be used to denote an anonymous list-variable. This last notation can improve the efficiency of list-matching because the instances of anonymous list-variables do not need to be built.

- annoted variable: the @ operator allows to give a variable name to a subterm. In f(x@g(_)) for example, x is a variable that will be instantiated by the instance of the subterm g(_). The variable x can then be used as any other variable.

- implicit notation: as explained below, the %op operator allows to give name to arguments. Assuming that the operator f has two arguments, named arg1 and arg2, then we can write the pattern f[arg1=a()] which is equivalent to f(a(),_). This notation can be interesting when using constructors with many subterms.

- unnamed list operator: it is often the case that given a list-sort, only one list-operator is defined. It this case, when there is no ambiguity, the name of the operator can be omitted. Considering the conc list-operator for example (see %oplist and %oparray below), to improve the readability, the pattern conc(_*,x,_*) can be written (_*,x,_*). This feature is particularly useful in the XML notation introduced in the following.

- symbol disjunction notation: to factorize the definition of pattern which have common subterms, it is possible to describe a family of patterns using a disjunction of symbols. The pattern (f|g)(a,b) corresponds to the disjunction f(a,b) or g(a,b). To be allowed in a disjunction (in standard notation), the constructors should have the same signature (arity, domain and codomain).

  In practice, it is usually better to use the disjunction notation with the implicit notation: ((f|g)[arg1=a()]). In that case, the signatures of symbols do not have to be identical: only involved slots have to be common (same names and types). Thus, the pattern (f|g)[arg1=a()] is correct, even if g has more slots than f: it only has to have the slot arg1, with the same sort.

  Note that the disjunction of symbol can also be used in XML notation: ¡a—b¿...¡/a—b¿.

3.4 XML pattern

To deal with XML documents, the XML notation can be used (<A><B attribute="name"/ ></A> for example).

When manipulating XML documents, we distinguish two main kinds of operations: retrieving information and transforming a document. Tom provides three different XML notations that ought to simplify the definition of patterns: the "standard " and the "implicit" XML notations are used to define compact (but incomplete) patterns. This notation is well suited to retrieve information. The "explicit" XML notation is used to precisely describe an XML pattern and all the variables that have to be instantiated. This notation is particularly well suited to perform XML transformation since it allows the programmer to precisely describe how variables have to be instantiated.

To make the XML notation understandable, we have to explain how XML documents are handled by Tom. To each XML document corresponds a DOM (Document Object Model) representation. In Tom, we have defined a mapping from DOM sorts to abstract algebraic sorts: TNode and TNodeList, which correspond respectively to Node and NodeList, defined by the Java DOM implementation.

Thus, a Node object becomes a ternary operator Element whose first subterm is the name of the XML node, the second subterm is a list of attributes and the third subterm is a list of subterms (which correspond to XML sub-elements). The second and the third elements are terms of sort TNodeList (because they are implemented by NodeList objects in DOM).

Thus, when considering the <A></A> XML document, the corresponding algebraic term is Element("A",[],[]), where [] denotes the empty list. Similarly, <A><Battribute="name"></B></A> is encoded into Element("A",[],[Element("B",[Attribute("attribute","name")],[])]).

When defining an XML pattern, the user has to introduce extra list-variables to precisely describe the XML pattern and capture the different contexts. Suppose that we are interested in finding a node <B></B> which is a subterm of a node <A></A> (but not necessary the first subterm). The algebraic pattern should be Element("A",[_*],[_*,Element("B",[_*],[_*]),_*]). Using the XML notation, this pattern

should be expressed as follows: <A(_*)>(_*,<B(_*)>(_*)</B>,_*)</A>. This notation (called explicit) is precise but error prone. This is why we have introduced the explicit notation, where all context variable can be removed (and () are replaced by []): <A[]><B[]>[]</B></A>. The last notation (called standard XML notation) allows the user to remove the [] and replace the list-separator (,) by spaces. The previous pattern can be written: <A><B></B></A>.

These three different notations allow the user to choose the level of control he wants to have on the XML pattern matching algorithm.

The formal description of the syntax is the following:

XMLTerm	::=	'<' *XMLNameList XMLAttributeList* '/>'
	|	'<' *XMLNameList XMLAttributeList* '>' *XMLChilds* '</' *XMLNameList* '>'
	|	'#TEXT' '(' *Identifier* | *String* ')'
	|	'#COMMENT' '(' *Identifier* | *String* ')'
	|	'#PROCESSING-INSTRUCTION' '(' (*Identifier* | *String*) ',' (*Identifier* | *String*) ')'
XMLNameList	::=	*XMLName*
	|	'(' *XMLName* ('|' *XMLName*)* ')'
XMLAttributeList	::=	'[' [*XMLAttribute* (',' *XMLAttribute*)*] ']'
	|	'(' [*XMLAttribute* (',' *XMLAttribute*)*] ')'
	|	(*XMLAttribute*)*
XMLAttribute	::=	'_*'
	|	*VariableName* '*'
	|	*AttributeName* '=' [*AnnotedName* '@'] (*Identifier* | *String*)
	|	[*AnnotedName* '@'] '_' '=' [*AnnotedName* '@'] (*Identifier* | *String*)
XMLChilds	::=	(*Term*)*
	|	'[' *Term* (',' *Term*)* ']'

3.5 Rule construct

In Tom, we can also define a set of rewrite rules. All the left-hand sides should begin with the same root symbol:

RuleConstruct	::=	'%rule' '{' (*Rule*)* '}'
Rule	::=	*RuleBody* (*RuleCondition*)*
RuleBody	::=	*Term* '->' *Term*
RuleCondition	::=	'where' *Term* ':=' *Term*
	|	'if' *Term* '==' *Term*

The %rule construct is composed of a list of conditional rewrite rules (the left-hand side is a term and the right-hand side is a term). All these rules (enclosed into a %rule { ... } construct) should begin with the same root symbol. The Tom compiler will generate a function (with a number of arguments equals to the arity of this root symbol) whose name corresponds to the name of this unique root symbol. Given a ground term, applying this function returns the instantiated and normalized right-hand side of the first rule (from top to bottom) whose pattern matches the considered subject and whose conditions are satisfied. When no rule can be applied (i.e. no pattern matches the subject, or no condition is satisfied), the given ground term, rooted by the root symbol of the rewrite system is returned.

In Tom, we consider two kinds of conditions:

- an equality condition ($t_1 == t_2$) is a pair of ground terms that belong to the same type. The condition is satisfied when the normal forms of the two terms (t_1 and t_2) are equal modulo the equals predicate defined in the definition of the type associated to t_1 and t_2.
- a matching condition ($p := t$) is a pair of terms where p may contains free variable. The condition is satisfied if the pattern p can be matched against the normal form of t. In this case, the free variables of p are instantiated and can be used in other conditions or the right-hand side of the rule.

3.6 Backquote construct

Another construct of Tom is the backquote (‘).

This construct can be used to build an algebraic term or to retrieve the value of a Tom variable (a variable instantiated by pattern-matching). The syntax of this operator is not fixed since it depends on the underlying language.

However, a backquote term should be of the following form:

- ‘*Name*: to denote a Tom variable
- ‘*Name*‘*’: to denote a Tom list-variable
- ‘*Name*(...): to build a prefix term
- ‘(...): to build an expression
- ‘*xml*(...): to build an XML term

In general, it is sufficient to add a backquote before the term you want to build to have the wanted behavior. The execution of ‘f(g(a)) will build the term $f(g(a))$, assuming that f, g, and a are Tom operators. Suppose now that g is no longer a constructor but a function of the host-language. The construction ‘f(g(a)) is still valid, but the semantic is the following: the constant a is built, then the function g is called, and the returned is put under the constructor f. Therefore, the result of g must be a correct term, which belongs to the right type (i.e. the domain of f).

To simplify the interaction with the host-language, it is also possible to use “unknown symbols” like f(x.g() or f(1+x). The scope of the backquote construct is determined by the scope of the most external enclosing braces, except in two case: ‘x and ‘x* which allows you to use variables instantiated by the pattern part. In that case the scope is limited to the length of the variable name, eventually extended by the ‘’*’’. Sometime, when writing complex expression like if(‘x==‘y —— ‘x==‘z), it can be useful to introduce extra braces (if(‘(x==y —— x==z))) to extend the scope of the backquote.

4 Tom signature constructs

4.1 Sort definition constructs

To define the mapping between the algebraic constructors and their concrete implementation, Tom provides a signature-mapping mechanism composed of several constructs. In addition to predefined mapping for usual builtin sorts (int, long, double, boolean, string, and char), all other algebraic sorts have to be declared using the %typeterm construct.

To use predefined sorts (in a %match construct or in the definition of a new operator), it is sufficient to use the %include construct (%include{ int.tom } for example).

When defining a new type with the %typeterm construct, two extra informations have to be provided:

- the implement construct describes how the new type is implemented. The host language part written between braces (’{’ and ’}’) is never parsed. It is used by the compiler to declare some functions and variables.
- the equals(t1,t2) construct corresponds to a predicate (parametrized by two term variables). This predicate should return true if the terms are “equal”. The true value should correspond to the builtin `true` value of the considered host language. This last optional predicate is used to compare builtin values and to compile non-linear left-hand sides.

Given a Java class Person we can define an algebraic mapping for this class:

```
%typeterm TomPerson {
  implement { Person }
  equals(t1,t2) { t1.equals(t2) }
}
```

Here, we assume that the method equals implement a comparison function over instances of Person. Note also that we have used TomPerson to make a clear distinction between algebraic sorts (defined in Tom) and implementation sorts (defined in Java, via the use of classes). In practice, we usually use the same name to denote both the algebraic sort and the implementation sort.

The grammar is the following:

IncludeConstruct	::=	'`%include`' '{' *FileName* '}'
VasConstruct	::=	'`%vas`' '{' *VasGrammar* '}'
GoalLanguageBlock	::=	'{' *BlockList* '}'
KeywordImplement	::=	'`implement`' *GoalLanguageBlock*
KeywordEquals	::=	'`equals`' '`(`' *Name* '`,`' *Name* '`)`' *GoalLanguageBlock*
TypeTerm	::=	'`%typeterm`' *Type* '{' *KeywordImplement* [*KeywordEquals*] '}'

4.2 Constructor definition constructs

Once algebraic sorts are declared, Tom provides a mechanism to define sorted signatures for constructors using %op, %oplist or %oparray constructs. When defining a new symbol with the %op construct, the user should specify the name of the operator, its codomain, and its domain. The later one is defined by a list of pairs (slot-name, sort).

Let us consider again the class Person, and let us suppose that an instance of Person has two fields (name and age), we can define the following operator:

```
%op TomPerson person(name:String, age:int)
```

In this example, the algebraic operator person has two slots (name and age) respectively of sorts String and int, where String and int are pre-defined sorts.

In addition to the signature of an operator, several auxiliary functions have to be defined:

- The is_fsym(t) { predicate(t) } construct is used to check if a term t is rooted by the considered symbol. The `true` value should correspond to the builtin `true` value of the considered host language (`true` in Java or Caml, and something different from `0` in C for example).
- The make(t1,...,tn) construct is parametrized by several variables (i.e. that should correspond to the arity of the symbol). A call to this make function should return a term rooted by the considered symbol, where each subterm correspond to the terms given in arguments to the function. When defining a constant (i.e. an operator without argument, make can be defined without brace: make { ... }).
- The get_slot(slotName,t) construct has to be defined for all slots of the signature. The implementation of these constructs should be such that the corresponding subterm is returned.

Coming back to our example, checking if an object t is rooted by the symbol person can be done by checking that t is an instance of the class Person. Building a person can be done via the Java function new Person(...). Accessing to the slots name and age could be implemented by an access to the variables of the class Person. In practice, the following operator definition should work fine:

```
%op TomPerson person(name:String, age:int) {
  is_fsym(t)  { t instanceof Person }
  make(t1,t2) { new Person(t1,t2) }
  get_slot(name,t) { t.name } // assuming that 'name' is public
  get_slot(age,t)  { t.age  } // assuming that 'age' is public
}
```

When defining a new symbol with the %oplist construct, the user has to specify how the symbol is implemented. In addition, the user has to specify how a list can be built and accessed:

- the make_empty() construct should return an empty list.

- the make_insert(e,l) construct corresponds to a function parametrized by a list variable and a term variable. This function should return a new list `l'` where the element `e` has been inserted at the head of the list `l` (i.e. equals(get_head(l'),e) and equals(get_tail(l'),l) should be `true`).
- the get_head(l) function is parametrized by a list variable and should return the first element of the considered list.
- the get_tail(l) function is parametrized by a list variable and should return the tail of the considered list.
- the is_empty(l) constructs corresponds to a predicate parametrized by a list variable. This predicate should return `true` if the considered list contains no element.

Similarly, when defining a new symbol with the %oparray construct, the user has to specify how the symbol is implemented, how an array can be built, and accessed:

- the make_empty(n) construct should return a list of size `n`.
- the make_append(e,l) construct corresponds to a function parametrized by a list variable and a term variable.

  Warning: This function should return a list `l'` such that the element `e` is at the `n-th` position.
- the get_element(l,n) construct is parametrized by a list variable and an integer. This should correspond to a function that return the `n-th` element of the considered list `l`.
- the get_size(l) constructs corresponds to a function that returns the size of the considered list. By convention, an empty list contains `0` element.

The %oplist or %oparray is complex but not difficult to use. Let us consider the ArrayList Java class, and let us define a Tom mapping over this data-structure. The first thing to do consists in defining the sort for the elements and the sort for the list-structure:

```
%typeterm Object {
  implement     { Object        }
  equals(l1,l2) { l1.equals(l2) }
}
%typeterm TomList {
  implement     { ArrayList    }
  equals(l1,l2) { l1.equals(l2) }
}
```

Once defined the sorts, it becomes possible to define the list-operator TomList conc(Object*). This operator has a variadic arity: it takes several Object and returns a TomList.

```
%oparray TomList conc( Object* ) {
  is_fsym(t)       { t instanceof ArrayList }
  make_empty(n)    { new ArrayList(n)       }
  make_append(e,l) { myAdd(e,(ArrayList)l)   }
  get_element(l,n) { (Object)l.get(n)        }
  get_size(l)      { l.size()                }
}

private ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}
```

An auxiliary function myAdd is used since the make_append construct should return a new list. The get_element should return an element whose sort belongs to the domain (Object) in this example. Although not needed in this example, in general, a cast ((Object)l.get(n)) is needed.

The grammar for the mapping constructs is the following:

KeywordIsFsym ::= ‘`is_fsym`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordGetSlot ::= ‘`get_slot`’ ‘(’ *Name* ‘,’ *Name* ‘)’ *GoalLanguageBlock*
KeywordMake ::= ‘`make`’ [‘(’ *Name* (‘,’ *Name*)* ‘)’] *GoalLanguageBlock*
KeywordGetHead ::= ‘`get_head`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordGetTail ::= ‘`get_tail`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordIsEmpty ::= ‘`is_empty`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordMakeEmptyList ::= ‘`make_empty`’ [‘(’ ‘)’] *GoalLanguageBlock*
KeywordMakeInsert ::= ‘`make_insert`’ ‘(’ *Name* ‘,’ *Name* ‘)’ *GoalLanguageBlock*
KeywordGetElement ::= ‘`get_element`’ ‘(’ *Name* ‘,’ *Name* ‘)’ *GoalLanguageBlock*
KeywordGetSize ::= ‘`get_size`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordMakeEmptyArray ::= ‘`make_empty`’ ‘(’ *Name* ‘)’ *GoalLanguageBlock*
KeywordMakeAppend ::= ‘`make_append`’ ‘(’ *Name* ‘,’ *Name* ‘)’ *GoalLanguageBlock*

Operator ::= ‘`%op`’ *Type Name* [‘(’ *SlotName* ‘:’ *Type* (‘,’ *SlotName* ‘:’ *Type*)* ‘)’]
‘{’ *KeywordIsFsym* (*KeywordMake* | *KeywordGetSlot*)* ‘}’
OperatorList ::= ‘`%oplist`’ *Type Name* ‘(’ *Type* ‘`*`’ ‘)’
‘{’ *KeywordIsFsym* (*KeywordMakeEmptyList* | *KeywordMakeInsert* | *KeywordGetHead*
| *KeywordGetTail* | *KeywordIsEmpty*)* ‘}’
OperatorArray ::= ‘`%oparray`’ *Type Name* ‘(’ *Type* ‘`*`’ ‘)’
‘{’ *KeywordIsFsym* (*KeywordMakeEmptyArray* | *KeywordMakeAppend*
| *KeywordElement* | *KeywordGetSize*)* ‘}’

4.3 Predefined sorts and operators

As mentioned previously, the system comes with several predefined signature-mappings for C, Java and Caml. Among them, let us mention:

- int.tom (written 1, 2, 3, *etc.*)
- long.tom (written 1l or 1L, *etc.*)
- double.tom (using Java grammar for double)
- boolean.tom (true or false)
- char.tom (written 'a', 'b', *etc.*)
- string.tom (written "a", "ab", *etc.*)

These mappings define, for each builtin sort of the host language, an algebraic sort that can be used in a %match or a signature definition construct. Thus, builtin values, such as f(5), g('a') or h("foo") , can be use in patterns.

The string.tom mapping is interesting because it provides an associative operator (concString) which allows the programmer to consider a string as a list of characters. Thus, the string "foo" can be seen as the algebraic object concString('f','o','o'). By using this mapping, it becomes possible to perform pattern matching against the content of a string. The pattern concString('f',X*) will match any string which begins with the character 'f'. By using the unnamed-symbol capability, this pattern can be written: ('f',X*) instead of concString('f',X*).

To match any string which begins with the substring "fo", the corresponding pattern should be ('f','o',X*). To simplify the definitions of such patterns, Tom supports an exception which allows the programmer to write ('fo',X*) instead. Internally, the ill-formed character 'fo' is expanded into the list ('f','o'). Thus, the annotated notation y@'fo' is expanded into (y@'f',y@'o'), and thus may be dangerous to use in this case.

In addition to these mappings, several other predefined mappings come with the system:

- aterm.tom and atermlist.tom provide a mapping to the C and Java version of the ATerm Library
- list.tom provides a mapping to the builtin notion of List in Caml
- dom.tom provides a mapping to the Java version of the DOM Library