TOM Language Reference

Pierre-Etienne Moreau
(with Julien Guyon and Christophe Ringeissen)

December 8, 2003

This manual also exists in Postscript or pdf.

Abstract: TOM is a Pattern Matching Preprocessor that can be used to intregrate term rewriting facilities in an imperative language such as C and Java.

Information on TOM is available the TOM web page tom.loria.fr.

Contents

Part I

Installing and using the system

Chapter 1

Installing and using the system

This part of the manual gives basic information to get and install the system.

1.1 Simple installation procedure

In order to quickly install the last version of Tom, the best thing to do is the following:

- **install the eclipse plugin**

or

- **browse the Tom web page tom.loria.fr**
- **get the last jtom-bundle-version.jar distribution**
- **get the associated scripts**
- **customize your CLASSPATH and PATH**
- **this should be sufficient**

1.2 Requirements and download

The current version of Tom is available from the TOM web page. In this page, you will find two kinds of releases:

- The stable distribution corresponds to major releases of Tom;
- The "Cutting Edge" distribution corresponds to the last releases of Tom. This is a way for keeping your system in sync with the latest developments. Be warned–the cutting edge is not for everyone!

Depending on the distribution, Tom is avaible in different formats:

- Releases: is a way to get a major release as a software package;
- Bundles: is a way to get a major release of Tom, including all needed dependencies;
- Jar distributions: corresponds to compiled stable versions of Tom;
- Jar devel distributions: corresponds to compiled development versions of Tom;
- Daily distribution: is a snapshot of the source control system. This is a way to get the last version of a package for which all the regression tests succeed;
- CVS distribution: is a way to get the last source version of the system.

Compiling and runtime tools TOM is written in Java and TOM itself. You will need a Java compiler and interpreter to compile and run TOM.
We actually use Sun JDK 1.4.x tools for development and runtime execution.
Some tests have already been realized with gcj and Jikes without any particular issues.

Package dependencies TOM software depends on others packages:

- shared-objects
- JJTraveler
- aterm-java
- apigen

The following figure shows the dependencies order:
Such packages are available from:

- TOM web page: tom.loria.fr
- CWI package base Page: www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/PackageBase

1.3 Installation

Depending on the TOM package you download, here are the installation instructions.

1.3.1 Installing the bundle package

This is the simplest way to have quickly TOM running on your computer.

- Untar the package:

```
> tar xzf jtom-bundle-VERSION.tar.gz
```

- Download and unpack the required packages:

```
> ./collect.sh
```

- Configure, build and install:

```
> ./configure --help gives a complete list of available configuration parameters.
> ./configure <<configuration parameters>>
> gmake
```

Once all is done, you are ready using TOM using jtom command.

1.3.2 Installing Tom package

Installation process

1. Get and install the shared-objects package

```
a. Type ‘./configure --prefix=<SHARED-DIR>’
b. Type ‘make’
c. Type ‘make install’
```

2. Get and install the JJTraveler package

```
a. Type './configure --prefix=<JJT-DIR>'
b. Type 'make'
c. Type 'make install'
```

3. Get and install the aterm-java package

```
a. Type './configure --prefix=<ATERM-DIR>
                     --with-JJTraveler=<JJT-DIR>
                     --with-shared-objects=<SHARED-DIR>'
b. Type 'make'
c. Type 'make install'
```

4. Configure and install Tom package:

```
a. './configure --prefix=<INSTALLDIR>
                --with-shared-objects=<SHARED-DIR>
                --with-aterm-java=<ATERM-DIR>'
b. Type 'make' to compile the package.
c. Type 'make install' to install the package.
```

1.4 Command line Arguments

1.4.1 NAME

jtom - compile Tom file (.t) into specified target languages

1.4.2 SYNOPSIS

```
jtom [-hceVvioDCfWldpsO] [-I path] filename[.t]
```

1.4.3 DESCRIPTION

`--help | -h`	Show the help
`--cCode | -c`	Generate C code (default is Java)
`--eCode | -e`	Generate Eiffel code (default is Java)
`--version | -V`	Print the version of TOM
`--verbose | -v`	Set verbose mode on: give duration information on each compilation passes
`--intermediate | -i`	Generate intermediate files
`--noOutput | -o`	Do not generate code
`--noDeclaration | -D`	Do not generate code for declarations
`--doCompile | -C`	Start after type-checking (used after a compilation process with –intermediate option
`--noCheck | -f`	Do not realize checking phases
`--Wall`	Print all warnings
`--noWarning`	Do not print warning
`--lazyType | -l`	Use universal type
`--import <path> | -I`	Path for %include construct to find included files
`--pretty | -p`	Generate readable code with indentation
`--atermStat | -s`	Print internal ATerm statistics
`--optimize | -O`	Optimized generated code
`--static`	Generate static functions
`--debug`	Generate debug primitives
`--memory`	Add memory management while debugging (not active with list matching)

1.5 For developpers

CVS repository The latest developments of Tom are available from anonymous cvs at:
cvs -d :pserver:cvs@cvs-sop.inria.fr:/CVS/aircube checkout jtom

Tom Architecture In TOM directory architecture, two main branches can be underlined: 'stable' and 'src'... The 'stable' branch contains the stable material to build a running TOM compiler. The 'src' branch is used to make new developments. Once in 'src' directory, you can modify the sources. then,

1. Type 'make' to build a new system and use jtom.src to use and test the new system.
2. At 'src' directory level, type 'make bootstrap' to bootstrap and install a new system. If something goes wrong, re-install the stable version: 'cd ../../stable'; 'make install' Then, correct the 'src' directory
3. If you are happy from the result, type 'make bootinstall' to make the current version become the stable one. **Be aware that this action may be dangerous and cause serious problems to your Tom installation!**

Please send a mail to `Pierre-Etienne.Moreau@loria.fr` to propose your suggestions.

Additional tools for developments To simplify TOM development, an extra package called Apigen (available at CWI package base Page) is used to generate data structures used by TOM. All generated files can be found in 'jtom/adt' directory.

Part II

The system

Chapter 2

Tom core

This part of the manual gives basic information on Tom internals.

2.1 Introduction

The current version of Tom is written in Java+Tom itself. It reads the program to be compiled and builds an abstract syntax tree (AST) to represent the program. The compiler is made up of stages, each of which can be seen as a process that transforms the AST into a new one. After the last transformation, the AST reprents a program closed to an imperative program. The last stage of the compiler is a generation phase that transform the AST into a concrete program written in C or Java, depending on the chosen target language.

2.2 The front-end

The front-end is the part of the system that reads the input program and builds the associated abstract syntax tree. This parts contains three components:

- a parser: this stage reads a concrete program written in the target language embedded with some Tom constructs, and build a first AST.
- an expander: this stage performs some very simple transformations such as macro expansion.
- a type checker: this stage add some type information to the AST (all untyped vaiables become a typed variable for example).

Because Tom is language independant, parsing a Tom program is not so easy. A solution could consists in implementing a specialized parser for each supported target language (one for C+Tom, and another one for Java+Tom), but for simplicity, and to keep the Tom system as simple as possible, we decided to implement a common parser, slightly specialised for each considered target language. When considering C and Java, we noticed that it is only needed to know how to recognize a string, a comment and a block to be able to make the difference between a target language construct and a Tomconstruct.

The main idea consists in synchronising the parser on several characters such as '%','"', '{' and '}'. For this purpose we decided to use Javacc and the lexical-mode facilities. Basically, the parser can be in two differents modes: `TomConstruct` mode and `TargetLanguage` mode. When being in the `TargetLanguage` mode, the parser reads everything unless a Tom construct (beginning with a '%' character) is recognized. Of course, this construct should not be in a target language string or comment. This explain why it is needed to be able to recognized such target language constructs. Once a Tom construct is recognized, the parser is switched to the `TomConstruct` mode, and the considered construct can be easilly parsed. We should notice that a Tom construct can also contains a target language part (always between '{' and '}'). When parsing such a part, the parser first reads a '{', and then is looking for a corresponding '}': for each encountered '}', it has to know if the read expression is well parenthesed or not. This explains why it is needed to be able to recognize and count the target language open and close block commands.

2.3 The compiler

The compiler receives as input an AST that merely corresponds to the input Tom program (this is roughly a list of interleaved target language constructs and Tom constructs). The goal of the compiler consists in transforming this AST into a "simpler one": an AST that can be easilly translated into an imperative program (a C or a Java program for example).

The kernel of the compiler contains a procedure that transforms a set of patterns into a automaton that implements a matching algorithm corresponding to the considered set of patterns. This automaton is then compiled into abstract instructions of the form: `IfThenElse`, `Assign`, `ExitAction`, `ExecuteAction`, etc. With such an approach, the most complex part of the compilation process is completly independ from the chosen target language, and can easily be reused.

2.4 The back-end

The back-end takes the last form of AST as input and generates a concrete program written in the target language. This stage consists in translating abstract instructions (like `IfThenElse`) into concrete instructions (like `if(cond) { instList }`) in C or Java. With such an approach, allows us to easily add a new back-end for any new supported target language.

Part III

The language

Chapter 3

The core language

This document is intended as a reference manual for the Tom language. It lists the language constructs, and gives their precise syntax and informal semantics. It is by no means a tutorial introduction to the language: there is not a single example.

3.1 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.

3.2 Lexical conventions

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

3.3 Names

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

3.4 Tom syntax

A TOM program is a target language program (namely C or Java) embedded with some new preprocessor constructs such as `%typeterm`, `%op`, `%rule` and `%match`. TOM is a multi-languages preprocessor, so, its syntax depends from the target language syntax. But for simplicity, we will only present the syntax of its constructs and explain how they can be integrated into the target language. Basically, a TOM program is list of blocks, where each block is either a TOM construct, either 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 target language program. So we have:

```
Tom         ::=  BlockList
BlockList   ::=
               (   MatchConstruct
               |   RuleConstruct
               |   BackQuoteTerm
               |   IncludeConstruct
               |   LocalVariableConstruct
               |   Operator
               |   OperatorList
               |   OperatorArray
               |   TypeTerm
               |   TypeInt
               |   TypeDouble
               |   TypeString
               |   TypeList
               |   TypeArray
               |   { BlockList }
               |   Other
               )*
```

- a *MatchConstruct* is translated into a list of instructions. This constructs may appear anywhere a list of instructions is valid in the target language.
- a *RuleConstruct* is translated into a function definition. This constructs may appear anywhere function declaration is valid in the target language.
- a *BackQuoteTerm* is translated into a function call.
- a *IncludeConstruct* is replaced by the content of the file referenced by the construct.
- a *LocalVariableConstruct* is replaced by variable declarations.
- *Operator*, *OperatorList* and *OperatorArray* are replaced by some functions definitions.
- *TypeTerm*, *TypeInt*, *TypeDouble*, *TypeString*, *TypeList* and *TypeArray* are also replaced by some functions definitions.

A `%match` construct contains two parts:

- a list of target language variables. These variables should contains the object on which patterns are matched
- a list of rules: a pattern and a semantic action (written in the target language)

The construct is defined as follow:

```
MatchConstruct    ::=  %match ( MatchArguments ) { ( PatternAction )* }
MatchArguments    ::=  Type SubjectName ( , Type SubjectName )*
PatternAction     ::=  MatchPatterns -> { BlockList }
MatchPatterns     ::=  Term ( , Term )*
```

A term has the following syntax:

```
Term              ::=  [ AnnotedName @ ] PlainTerm
PlainTerm         ::=  XMLTerm
                     |  VariableName *
                     |  HeadSymbolList [ ExplicitTermList | ImplicitPairList ]
                     |  ExplicitTermList
                     |  _
                     |  _*
HeadSymbolList    ::=  HeadSymbol
                     |  ( HeadSymbol ( | HeadSymbol )+ )
ExplicitTermList  ::=  ( Term ( , Term )* )
ImplicitPairList  ::=  [ PairTerm ( , PairTerm )* ]
PairTerm          ::=  [ SlotName = Term ]
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)* ]
XMLAttribute      ::=  AttributeName = Term
                     |  _*
                     |  VariableName *
                     |  [AnnotedName] @] _ = Term
XMLChilds         ::=  ImplicitTermList
                     |  ( Term )*
ImplicitTermList  ::=  [ Term ( , Term )* ]
```

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
```

```
IncludeConstruct  ::=  %include { FileName }
Operator          ::=  %op Type Name [ ( [ SlotName : ] Type ( , [ SlotName : ] Type )* ) ]
                       { KeywordFsym ( KeywordMake | KeywordGetSlot | KeywordIsFsym)* }
OperatorList      ::=  %oplist Type Name ( Type * )
                       { KeywordFsym ( KeywordMakeEmptyList | KeywordMakeInsert )* }
OperatorArray     ::=  %oparray Type Name ( Type * )
                       { KeywordFsym ( KeywordMakeEmptyArray | KeywordMakeAppend )* }
```

```
TypeTerm   ::=  %type Type { KeywordImplement ( KeywordGetFunSym | KeywordGetSubterm | KeywordCmpFunS
TypeList   ::=  %typelist Type { KeywordImplement [KeywordGetFunSym] [KeywordGetSubterm] [KeywordCmpF
TypeArray  ::=  %typearray Type { KeywordImplement ( KeywordGetFunSym | KeywordGetSubterm | KeywordCm
```

GoalLanguageBlock ::= { *BlockList* }
KeywordImplement ::= `implement` *GoalLanguageBlock*
KeywordGetFunSym ::= `get_fun_sym (` *Name* `)` *GoalLanguageBlock*
KeywordGetSubterm ::= `get_subterm (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordCmpFunSym ::= `cmp_fun_sym (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordEquals ::= `equals (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordGetHead ::= `get_head (` *Name* `)` *GoalLanguageBlock*
KeywordGetTail ::= `get_tail (` *Name* `)` *GoalLanguageBlock*
KeywordIsEmpty ::= `is_empty (` *Name* `)` *GoalLanguageBlock*
KeywordGetElement ::= `get_element (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordGetSize ::= `get_size (` *Name* `)` *GoalLanguageBlock*
KeywordFsym ::= `fsym` *GoalLanguageBlock*
KeywordMake ::= `make (` *Name* (`,` *Name*)* `)` *GoalLanguageBlock*
KeywordIsFsym ::= `is_fsym (` *Name*)* `)` *GoalLanguageBlock*
KeywordGetSlot ::= `get_slot (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordMakeEmptyList ::= `make_empty` [`( )`] *GoalLanguageBlock*
KeywordMakeInsert ::= `make_insert (` *Name* `,` *Name* `)` *GoalLanguageBlock*
KeywordMakeEmptyArray ::= `make_empty (` *Name* `)` *GoalLanguageBlock*
KeywordMakeAppend ::= `make_append (` *Name* `,` *Name* `)` *GoalLanguageBlock*

A last construct of Tom is the backquote (‘). This construct can be used to build an algebraic term. The syntax of this operator is not fixed since it depends on the underlying language.

However, a backquote term should at least contain a function call or the construction of a prefix term: ‘*Name*`(` ... `)`.

For building an XML term, the expression should begin with ‘*xml*`(` ... `)`.

3.5 Tom semantic

3.5.1 Type definition

When defining a new type with the `%typeterm` constructs, several access functions have to be defined:

- the `implement` construct describes how the new type is implemented. The target language part written between braces (`'{'` and `'}'`) is never parsed. It is used by the compiler to declare some functions and variables.
- the `get_fun_sym(t)` construct corresponds to a function (parameterized by a term variable) that should return the root symbol of a given term (the term referenced by the term variable `t` in this example).
- the `cmp_fun_sym(s1,s2)` construct corresponds to a predicate (parameterized by two symbol variables). This predicate should return `true` if the symbols are “equal”. The `true` value should correspond to the builtin `true` value of the considered target language. (`true` in `Java`, and something different from `0` in `C` for example).
- the `get_subterm(t,n)` construct corresponds to a function (parameterized by a term variable and an integer). This function should return the `n-th` subterm of the term `t`. This never called with and integer parameter that does not correspond to the arity of the root symbol of the considered term (i.e. we alway have $0 <= n < arity$).
- the `equals(t1,t2)` construct corresponds to a predicate (parameterized 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 target language. This last optional predicate is only used to compile non-linear left-hand sides. It is not needed, if the specification does not contain such patterns.

When defining a new type with the `%typelist` construct, several other access functions have to be defined:

- the get_head(l) function is parameterized by a list variable and should return the first element of the considered list.
- the get_tail(l) function is parameterized by a list variable and should return the tail of the considered list.
- the is_empty(l) constructs corresponds to a predicate parameterized by a list variable. This predicate should return `true` if the considered list contains no element.

When defining a new type with the %typearray construct, the two different access functions have to be defined:

- the get_element(l,n) construct is parameterized 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.

When introducing integer type with the %typeint construct, then the int type can be directly used as a term.

3.5.2 Operator definition

When defining a new symbol with the %op s construct, the user should specify how the symbol s is implemented. This is done by the fsym { implementation_of_s } construct. The expression between braces should correspond (modulo the cmp_fun_sym predicate) to the expression returned by the function get_fun_sym applied to a term rooted by the considered symbol (s in this case).

The isfysm(t) { predicate(t) } construct can be used to specialize this mechanism. When isfsym is defined, the predicate (given between braces) is used to check if a term t is rooted by the considered symbol (s in this example). This replaces the use of get_fun_sym and cmp_fun_sym.

When defining a symbol, is it also possible to specify a make construct. This function is parameterized by several term 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.
As mentioned in the syntax definition, it is also possible to name each field of a constructor symbol by using the `Type f(name1:Type, name2:Type2)` syntax. Adopting this programming style has two main advantages:

- when writing a pattern, this allows you write `f[name2=a]` instead of `f(_,a)`. One benefit is that you can modify the signature (adding a field for example) without necessary having to modify every pattern that occurs in the program.
- you may also specialize the get_subterm access function for a given constructor. This can be done with the get_slot construct.

When defining a new symbol with the %oplist construct, the user has to specify how the symbol is implemented. The user has also to specify how a list can be built (this is no longer optional as in the %op construct):

- the make_empty() construct should return an empty list.
- the make_insert(e,l) construct corresponds to a function parameterized 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`).

When defining a new symbol with the %oparray construct, the user has to specify how the symbol is implemented. The user has also to specify how a list can be built (this is no longer optional as in the %op construct):

- the make_empty(n) construct should return a list of size `n`.

- the `make_append(e,l)` construct corresponds to a function parameterized 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.

3.5.3 Match definition

In order to match patterns against a list of subjects, Tom provides the `%match` construct. This constructs contains two parts:

- a *MatchArguments*: this is a list of (target language) variables that reference the terms to be matched.

- a list of *PatternAction*: this is a list of pairs (pattern,action), where an action is a set of target language instructions.

The `%match` construct is evaluated in the following way:

- given a list of ground terms (referenced by the list of target language variables), the execution control is transfered to the first *PatternAction* whose patterns match the list of ground terms.

- given a *PatternAction*, the list of free variables is instantiated and the associated semantic action is executed. If the execution control is transfered outside the `%match` instruction (by a `goto`, `break` or `return` for example), the matching process is finished. Otherwise, it is continued as follows:

  - if the considered matching theory may return several matches (list-matching for instance), for each match, the list of free variables is instantiated and the associated semantic action is executed.

  - when all matches have been computed (there is at most one match in the syntactic theory), the execution control is transfered to the next *PatternAction* whose patterns match the list of ground terms.

- when there is no more *PatternAction* whose patterns match the list of ground terms, the `%match` instruction is finished, and the execution control is transfered to the next instruction.

Note: the behavior is not determined if a semantic action modifies a target language variable which is an argument of a `%match` instruction under evaluation.

3.5.4 Rule definition

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 should begin with the same root symbol. The Tom compiler should generate a function (with one argument) whose name correspond to the name of this unique root symbol. Given a ground term, applying this function returns the instantiated right-hand side of the first rule 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.

3.5.5 Pattern definition

A pattern is a term which could contain variables. These variables will be instantiated by the matching procedure. In Tom, the variables do not have to be declared: their type is inferred automatically, depending on the context in which the 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`) 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.

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

- unamed list operator: it is not rare 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, to improve the readability, the pattern `conc(_*,x,_*)` can be written `(_*,x,_*)`.

- 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)` correspond to the disjunction `f(a,b)` or `g(a,b)`. To be allowed in a disjunction, the constructor should have the same signature (arity, domain and codomain). The disjunction notation can be used with the implicit notation: `(f|g)[arg1=a()]`.

3.5.6 XML pattern

there exist several ways to define an XML pattern. The simplest one consist in using the "standard " XML notation. In Tom, we have decided to compile XML pattern matching by using associative list-matching. Thus, to precisely describe XML patterns, we need to introduce extra list-variables which should capture the context. This can be done by using anonymous list-variables `_*`. However, to simplify the definition of XML patterns we have also introduced an implicit notation which automatically adds these variables. An XML term is a term of sort `TNode`. As described in the ADT file, an `ElementNode` has 3 subterms: a name, a list of attributes and a list a subterms.

- explicit notation:

- implicit notation:

- prefix notation:

Part IV

Libraries

Chapter 4

Runtime libraries

4.1 Runtime and traversal functions

4.2 Debug

Part V

Tutorial

Chapter 5

Tutorial and examples

On top of TOM directory, you can find the tutorial directory where all examples presented here can be found and ran. The difficulties encountered in this tutorial is increasing with each section and subsection. Most of the examples are based on Java and the corresponding ATerm library for terms representation. Each example can be ran using this default suite of commands:

```
> jtom Example.t
> javac Example.java
> java Example
```

Extra commands are explain when necessary.

5.1 Introduction to the ATerm library for Java

ATerm is an abstract data type designed for tree-like data structures representations. More information on the C version is available at: http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary.

5.2 Peano integer

The Peano integer formalism aims to represent integer as Zero or as the successor of an integer Suc(int).

5.2.1 Simple examples

Terms definition TOM allows to define such algebraic specifications. First, we define the sort peano and define the 2 operators: zero and suc.

```
%typeterm peano {


    implement{ ATerm }
    get_fun_sym(t) { (((ATermAppl)t).getAFun()) }
    cmp_fun_sym(t1,t2) { t1 == t2 }
    get_subterm(t, n) { (((ATermAppl)t).getArgument(n)) }


}

%op peano zero {


    fsym{ fzero /*factory.makeAFun("zero",0,false*/}
```

```
}

%op peano suc(term) {

    fsym{ fsuc /*fsuc = factory.makeAFun("suc",1,false)*/}

}
```

This is the minimal material given to TOM to compile pattern matching constructs on peano sort. In particular, the sort is map to ATerm java class (see `implement`). TOM now knows the functional symbol of zero and suc, how to get and compare them;

Here is the code generated by TOM:

```
public Object tom_get_fun_sym_peano(ATerm t) { return (((ATermAppl)t).getAFun()) ; }
public Boolean tom_cmp_fun_sym_peano(Object t1, Object t2) { return t1 == t2 ; }
public Object tom_get_subterm_peano(ATerm t, int n) { return (((ATermAppl)t).getArgument(n)) ; }
```

Using match structure to make pattern matching Once the operators are defined, let us consider a simple symbolic computation (addition) defined on Peano integers.

```
public ATerm plus(ATerm t1, ATerm t2) {

    %match (peano t1, peano t2) {

        x,zero -¿ { return x; }
        x,suc(y) -¿ { return suc(plus(x,y)); }

    }
    return null;

}
```

Note: We can easily see that this match program is complete (We match all possibilities) but we need to add a last return statement else Java compiler will complain.

How to run the example By running the command:
¿ jtom Peano1;javac Peano1.java;java Peano1
We get the result:
plus(10,10) = suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(zero))))))))))))))))))

How does this work The peano integer 10 is created by hand using the instruction for(int i=0 ; i¡n ; i++) { N = suc(N);}and the defined java suc function. The match construct signature takes 2 peano terms. The first pattern x, zero will be transformed in first a substitution then in a ifthenelse construct. Indeed, x represents a variable because it is not zero nor suc. So, x will be substitued by t1. Then, the compilation will make a test on t2 being a zero or not. The second pattern will be processed the them. Here you can see the result of the compilation in the target language, ie Java:

```
public ATerm plus(ATerm t1, ATerm t2) {

    ATerm tom_match1_1 = null;
    ATerm tom_match1_2 = null;
    tom_match1_1 = (ATerm) t1;
    tom_match1_2 = (ATerm) t2;
    matchlab_match1_pattern1: {
```

```
ATerm x = null; x = (ATerm) tom_match1_1; \*=¿substitution of x*\
if(tom_cmp_fun_sym_peano(tom_get_fun_sym_peano(tom_match1_2) , fzero)) \*=¿compilation
of first pattern*\{


    return x; \* =¿Action associated to the first pattern *\

}
matchlab_match1_pattern2: {
ATerm y = null; ATerm x = null; x = (ATerm) tom_match1_1;
if(tom_cmp_fun_sym_peano(tom_get_fun_sym_peano(tom_match1_2) , fsuc)) \*=¿compilation
of second pattern*\{


    ATerm tom_match1_2_1 = null; tom_match1_2_1 = (ATerm) tom_get_subterm_peano(tom_match1_2,
    0);
    y = (ATerm) tom_match1_2_1;
    return suc(plus(x,y));


}
}

return null;
}
```

The Peano2 examples shows how to map an algebraic sprcification to another representation in the target language. So, in this second example, you see that terms are implemented by **ATermAppl** instead of **ATerm**. This implementation choice will simplify the way `implement`, `get_fun_sym`, `cmp_fun_sym` and `get_subterm` are implemented (No more cast needed).

5.2.2 What about with C language

A corresponding example to Peano1 can be found: cpeano.t. To compile and run it, use:
¿jtom -c cexample.t
¿ gcc cexample.c

One of the major drawbcak in this example is the complexity of defining terms in regard to previous examples using ATerm for Java. A second example, cpeanofib, gives a more complex implementaion of terms. Of course, this is still simple in regard to the low complexity of represented terms.

5.2.3 Using Bacquote construct

The Peano3 examples illustrates the way the backquote construct can be used to simplify term creation. This is realized by adding a make function to each defined operator. This way, the Java suc function is no more usefull and is directly replace by the corresponding `make` in suc operator definition.

```
%op term zero {


    fsym{ factory.makeAFun("zero",0,false) }
    make{ factory.makeAppl(factory.makeAFun("zero",0,false))}


}

%op term suc(term) {
```

```
    fsym factory.makeAFun("suc",1,false)
    make(t)  factory.makeAppl(factory.makeAFun("suc",1,false),t)


}
```

Then, the creation of the term representing the integer N become easy:
ATermAppl N = ‘zero(); for(int i=0 ; i¡n ; i++) {
N = ‘suc(N);
}

But, the Backquote construct is more powerfull. Indeed, it is translated in a function call. This allows to write in the macth construct:
x,suc(y) -¿ return ‘suc(plus(x,y));
The plus function we be called and its result will be passed to create the successor term; Of course, this is correct because plus returns a term expected by the successor operator: suc. Running TOM on this example leeds for a warning message saying that it can not inferate type-checking for this construct, because it has no knowledge on plus function. To avoid the warning messages, please use:
¿ jtom -W Peano3

5.2.4 Advanced examples and more powerfull operators

As we see in previous examples, the operator definition can be developped to simplify the use of TOM main construct. We can also supply others functions to specialized their default behavior defined in the corresponding term definition.
The Peano1 example shows this point:

```
%typetermterm {


    implement{ ATermAppl }
    get_fun_sym(t) { null }
    cmp_fun_sym(t1,t2) { false }
    get_subterm(t, n) { null }


}

%opterm zero {


    fsym{ /* empty */ }
    is_fsym(t) { t.getAFun() == factory.makeAFun("zero",0,false) }
    make{ factory.makeAppl(factory.makeAFun("zero",0,false)) }


}

%opterm suc(pred:term) {


    fsym{ /* empty */ }
    is_fsym(t) { t.getAFun() == factory.makeAFun("suc",1,false) }
    get_slot(pred,t) { (ATermAppl)t.getArgument(0) }
    make(t) { factory.makeAppl(factory.makeAFun("suc",1,false),t) }


}
```

In the Peano2 example, we can see the facilities offered to extract slot from match construct but also from term using defined slot names or facilities.

```
public ATermAppl fib(ATermAppl t) {

  %match(term t) {

    y@zero -¿ { return ‘suc( y); }
    y@suc(zero) -¿ { return y; }
    suc[pred=y@suc(x)] -¿ { return ‘plus(fib(x),fib(y)); }

  }
  return null;

}
```

5.3 Integer and Fibonacci

As seen in previous examples, all results leed to expression of the result in Peano paradigm. TOM offers a way to directly works with integer.

```
public class Jint {

  %typeint
  public final static void main(String[] args) {
  Jint test = new Jint();
  int res = test.fib(10);
  System.out.println("res = " + res);
  }
  public int fib(int t) {

    %match(int t) {
    0 -¿ { return 1; }
    1 -¿ { return 1; }
    n -¿ { return fib(n-1) + fib(n-2); }
    }

  }

}
```

5.4 List

TOM allows to work on list representation of terms. 2 examples are available to show how to use either `%typelist` either `%typearray` and their associated creator operator: `%oplist` and `%oparray`.

5.4.1 Defining List

```
%typelistTomList {

    implement{ ATermList }
    get_fun_sym(t) { ((t instanceof ATermList)?factory.makeAFun("conc", 1, false):null) }
    cmp_fun_sym(t1,t2) { t1 == t2 }
    equals(l1,l2) { l1==l2 }
    get_head(l) { l.getFirst() }
    get_tail(l) { l.getNext() }
    is_empty(l) { l.isEmpty() }

}

  %oplistTomList conc( TomTerm* ) {

    fsym{ factory.makeAFun("conc", 1, false) }
    make_empty() { factory.makeList() }
    make_insert(e,l) { l.insert(e) }

}
```

5.4.2 Defining Array

```
%typearrayTomArray {
implement{ ArrayList }

    get_fun_sym(t) { ((t instanceof ArrayList)?factory.makeAFun("conc", 1, false):null) }
    cmp_fun_sym(t1,t2) { t1 == t2 }
    equals(l1,l2) { l1.equals(l2) }
    get_element(l,n) { l.get(n) }
    get_size(l) { l.size() }

}
%oparrayTomArray conc( TomTerm* ) {

    fsym{ factory.makeAFun("conc", 1, false) }
    make_empty(n) { myEmpty(n) }
    make_append(e,l) { myAdd(e,(ArrayList)l) }

}
```

5.4.3 Swapsort and the power of Tom

The 2 examples are very similar and allow to define 2 functions: swapSort and removeDouble ina very efficient way.
Have a look to swapSort function definition in List2.t (with array):

```
  public ArrayList swapSort(ArrayList l) {
%match(TomList l) {
```

```
    conc(X1*,x,X2*,y,X3*) -¿ {

        String xname = x.getName();String yname = y.getName();
        if(xname.compareTo(yname) ¿ 0)  return ‘swapSort(conc(X1*,y,X2*,x,X3*));

    }

}
```

The compilation of the Match construct allows to generate all combinations on the list parameter until it found the entered pattern. Using a Backquote construct allows to naturally recursively call the defined function!!!!!

5.5 Apigen and the automatic generation of term representation

One of the more difficult stuff in writting Tom program is to write all stuff concerning term definitions. To help developpers, a usefull tools called Apigen is available at CWI package base Page and use by Tom developpers. This tools take an abstract data type file called ADT file and generate C or Java data-types that hide the ATerm library behind a typed API. For Tom, it generates more than 300 java files to defining all used Tom structures. For installing Apigen, see the developpers sections on install manual.

Both Java and C examples have an associated README file where you can found compilation instructions.

5.5.1 Description of Apigen

Full documentation on Apigen can be found at http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ApiGen.

5.5.2 Java example

All generated file

5.5.3 C example

5.6 Tom runtime and Mathematical expression

This last example combines all previous items in a single problem: Derivation and simplification of mathematical expressions. But, it also introduces the way the runtime library can be used.

5.7 Debugging Tom