Touist reference manual

2017-03-29 (touist v3.0.0-2-g3cbfad9)

Olivier Gasquet

Frédéric Maris

Maël Valais

Contents

1. Introduction

Touist is a language that allows to express propositional logic [1, 2]. You are provided with two programs: a graphical interface, refered as `touist.jar` (as it is written in Java) and `touist`, the command-line compiler and solver (written in Ocaml).

The touist language aims at making the writing of propositional logic as approchable as possible. Propositions and connectors syntaxes are close to what would be written on paper. Propositions are words that can contain numbers, the character "`_`". Here are some examples of formulas:

Propositional logic	**`touist` language**
$\neg p$	`not p`
$p \wedge q$	`p and q`
$p \vee q$	`p or q`
$p \oplus q$	`p xor q`
$p \rightarrow q$	`p => q`
$p \leftrightarrow q$	`p <=> q`

After typing a formula, you can ask `touist` to find a valuation (true or false) of each proposition so that the whole formula is true (such valuation, also called *interpretation*, is called *model*). When a model exists, your formula is *satisfiable*. For example, a model of $p \vee q$ is $\{p = true, q = false\}$. To check the models of this formula using `touist`, you can do

Graphical Java interface	**Command-line interface**[1]
1. Type `p and q`	1. Create a file `p and q`
2. Press "Solve"	2. Type `./touist ---sat --solve yourfile`
3. Press "Next" to see other models	3. The first model is displayed

- –sat –solve‘

1.1. Simple inference checking

Checking the truth of a given reasoning (or *inference*) may be handy. From a wikipedia example:

Premise 1:	If it's raining then it's cloudy
Premise 2:	It's raining
Conclusion:	It's cloudy

This inference can be written

$$\{raining \rightarrow cloudy, raining\} \models cloudy$$

The *infer* symbol ($\models$) does not belong to the `touist` language (we call it "meta-language"). This means that we have to transform this notation to an actual propositional formula.

Theorem 1.
Let H be a set of formulas (called *hypotheses* or *premises*) and C a formula (called *conclusion*). Then

$$H \models C \text{ if and only if } H \cup \{\neg C\} \text{ is unsatisfiable.}$$

From this theorem, we just have to check that the set of formulas

$$\{raining \to cloudy, raining, \neg cloudy\}$$

has no model. We can translate this set to `touist` language (comments begin with two semi-colon "`;;`"):

```
raining => cloudy       ;; Premise 1
raining                 ;; Premise 2
not cloudy              ;; Conclusion
```

Note. In touist, the premises are simply formulas separated by a new line. A new line is semantically equivalent to the `and` connector: the previous bit of `touist` code could be equivalently written

```
(raining => cloudy) and raining and not cloudy
```

2. Language reference

2.1. Structure of a touist file

```
<touist-file> ::= <affect> <touist-file>
                | <formula> <touist-file>
                | EOF
```

A touist file is a whitespace-separated list of affectations and formulas. Affectations are global and cannot be done in nested formulas. They can be anywhere in the file (at the beginning, at the end or interlaced with formulas). A whitespace is a space, tab or newline.

Comments begin with the "`;;`" sequence.

2.2. Affectations

Affectations have the following syntax:

```
<affect> ::= <var> "=" (<int>|<float>|<bool>|<prop>|<set>)
```

This kind of affectation is global and applies for the whole code, even if the affectation happens after the formulas when the variable appears. These affectations are evaluated before any formula.

The only place where the order of affectation has importance is when you want to use a variable in an affectation expression. For example:

```
$N = 10
$set = [1..$N]    ;; $N must be defined before $set
```

2.3. Variables

```
TERM = [_0-9]*[a-zA-Z][a-zA-Z_0-9]*
<var> ::=
  | "$" TERM                              (* simple-var *)
  | "$" TERM "(" <comma-list(<float>|<int>|<prop>)> ")" (* tuple-var *)
```

Simple variable ("simple-var")

A simple variable is of the form `$my_var`. In a formula, a simple variable is always expected to be a proposition. In an expression, a simple variable can contain an integer, a floating-point, a proposition, a boolean or a set.

Tuple variable (can be seen as a *predicate*)

A tuple variable is a simple variable followed by a comma-separated list of indices in braces, e.g., `$var($i,a,4)`. The leading variable (`$var`) must always contain a proposition. The nested indices (e.g., `$i`) can be integers, floats, propositions or booleans.

A tuple variable will always be expanded to a proposition. For example, if `$var=p` and `$i=q`, then it will expand to `p(q,a,4)`

Tuple variables are not (yet) compatible with the set-builder construct (in 2.7.3). If one of the indices is a set, the set will stay as-is.

Here are some examples of variables:

Simple-var	Tuple-var
`$N`	`$place($number)`
`$time`	`$action($i,$j)`
`$SIZE`	
`$is_over`	

2.4. Propositions

```
TERM = [_0-9]*[a-zA-Z][a-zA-Z_0-9]*
<prop> ::=
    | <var>
    | TERM
    | TERM "(" <comma-list(<float> | <int> | <prop>)> ")"
```

A simple proposition is a simple word that can contain numbers and the underscore symbol ("_"). A tuple proposition (we can it as a *predicate*), of the form `prop(1,$i,abc)`, must have indices of type integer, float, boolean or set.

2.4.1. Tuple proposition containing a set

A tuple proposition that is in an expression and that contains at least one set in its indices will be expanded to a set of the cartesian product of the set indices. This feature is called **set-building** and is described in 2.7.3 and only works in expressions (not in formulas).

In the following table, the two right-columns show how the propositions are expanded whether they are in an expression or in a formula:

Proposition	is in a formula	is in an expression
`p([a])`	`p([a])`	`p(a)`
`p([a,b,c])`	`p([a,b,c])`	`[p(a),p(b),p(c)]`
`p([a,b],[1..2])`	`p([a,b],[1..2])`	`[p(a,1),p(b,1)` `p(a,2),p(b,2)]`

2.5. Numeric expression

The available operations on integers and floats are `+`, `-`, `*`, `/`, `$x mod $y` (modulo) and `abs($x)` (absolute value).

Here is the complete rule for numeric operators:

```
<num-operation(<T>)> ::=
    | <T> "+" <T>
    | <T> "-" <T>
    |     "-" <T>
    | <T> "*" <T>
    | <T> "/" <T>
<num-operation-others(<T>)> ::=
    | <T> "mod" <T>
    | "abs(" <T> ")"
```

Note. Integer and float expressions cannot be mixed. It is necessary to cast explicitely to the other type when the types are not matching. For example, the expression `1+2.0` is invalid and should be written `1+int(2.0)` (gives an integer) or `float(1)+2.0` (gives a float). Some operators are specific to integer or float types:

- `card([a,b])` returns an integer,
- `sqrt(3)` returns a float.

2.5.1. Integers

An integer constant `INT` is a number that satisfies the regular expression `[0-9]+`. Here is the rule for writting correct integer expressions:

```
<int> ::=
    | "(" <int> ")"
    | <var>
    | INT
    | num-operation(<int>)
    | num-operation-others(<int>)
    | "if" <bool> "then" <int> "else" <int> "end"
    | "int(" (<int>|<float>) ")"
    | "card(" <set> ")"
```

2.5.2. Floats

A floating-point constant FLOAT is a number that satisfies the regular expression `[0-9]+\.[0-9]+`. The variants `1.` or `.1` are not accepted. Here is the rule for writting correct integer expressions:

```
<float> ::=
    | "(" <float> ")"
    | <var>
    | FLOAT
    | num-operation(<float>)
    | num-operation-others(<float>)
    | "if" <bool> "then" <float> "else" <float> "end"
    | "float(" (<int>|<float>)  ")"
    | "sqrt(" <float> ")"
```

2.6. Booleans

The constants are `true` and `false`. The boolean connectors are $>$, $<$, $\geq$ (`>=`), $\leq$ (`<=`), $=$ (`==`) and $\neq$ (`!=`). The operators that return a boolean are `subset($P,$Q)`, `empty($P)` and `p in $P`:

`subset($P,$Q)`	$P \subseteq Q$	P is a subset (or is included in) Q
`empty($P)`	$P = \emptyset$	P is an empty set
`$i in $P`	$i \in P$	i is an element of the set P

Sets are detailed in 2.7.

Note. Booleans cannot be mixed with formulas. In a formula, the evaluation (choosing true or false) is not done during the translation from touist to the "solver-friendly" language. Conversely, a boolean expression must be evaluable during the translation.

Here is the full grammar rule for booleans:

```
<bool> ::= "(" <bool> ")"
    | <var>
    | "true"
    | "false"
    | (<int>|<float>|<prop>|<bool>) "in" <set>
    | "subset(" <set> "," <set> ")"
    | "empty(" <set> ")"
    | <equality(<int>|<float>|<prop>)>
    | <order(<int>|<float>)>
    | <connectors(<bool>)>
<equality(<T>)> ::=
    | <T> "!=" <T>
    | <T> "==" <T>
<order(<T>)> ::=
    | <T> ">" <T>
```

```
| <T> "<" <T>
| <T> "<=" <T>
| <T> ">=" <T>
```

2.7. Sets

Sets can contain anything (propositions, integers, floats, booleans or even other sets) as long as all elements have the same type. There exists three ways of creating a set:

2.7.1. Sets defined by enumeration

$\{1, 3, 8, 10\}$ can be written `[1,2,3]`. Elements can be integers, floats, propositions, booleans or sets (or a variable of these five types). The empty set $\emptyset$ is denoted by `[]`.

2.7.2. Sets defined by a range

$\{i \mid i = 1, \ldots, 10\}$ can be written `[1..10]`. Ranges can be produced with both integer and float limits. For both integer and float limits, the step is 1 (it is not possible to change the step for now);

2.7.3. Set-builder notation

$\{p(x_1, ..., x_n) \mid (x_1, ..., x_n) \in S_1 \times ... \times S_n\}$ is the set of tuple propositions based on the cartesian product of the sets $S_1, ..., S_n$. This can be written using sets instead of atoms in a proposition: `p([a,b,c])` will produce `[p(a),p(b),p(c)]`. You can mix sets, integers, floats, propositions and booleans in indices:

Proposition	produces the set
`f(1,[a,b],[1.0..2.0])`	`[f(1,a,1.0),f(1,a,2.0),` `f(1,b,1.0),f(1,b,2.0)]`

Important: the set-builder feature only works in expressions and does not work in formulas. In formulas, the proposition `f([a,b])` will simply produce `f([a,b])`. This also means that you can debug your sets by simply putting your set in a tuple proposition.

This notation is inspired from the concept of extension of a predicate (cf. wikipedia).

2.7.4. Operators using sets

Four operators have been mentionned in previous sections; refer to these sections to get description of each:

- `card(x)` (cf. 2.5.1),
- `empty($S)` (cf. 2.6),
- `1 in [1,2,3]` (cf. 2.6),

- `subset($P,$Q)` (cf. 2.6)

Along with these four operators, some common set operators are available. Let P and Q denote two sets:

`inter($P,$Q)`	$P \cap Q$	intersection
`union($P,$Q)`	$P \cup Q$	union
`diff($P,$Q)`	$P \setminus Q$	difference
`powerset($Q)`	$\mathcal{P}(Q)$	powerset (cf. 2.7.4.1)

Powerset The `powerset($Q)` operator generates all possible subsets S such that $S \subseteq Q$. It is defined as

$$\mathcal{P}(Q) := \{S \mid S \subseteq Q\}$$

The empty set is included in these subsets. Example: `powerset([1,2])` generates `[[],[1],[2],[1,2]]`.

Here is the complete rule for sets:

```
<set> ::= "(" <set> ")"
    | <var>
    | "[" <comma-list(<int>|<float>|<prop>|<bool>)> "]"
    | "[ <int> ".." <int> "]"
    | "union(" <set> "," <set> ")"
    | "inter(" <set> "," <set> ")"
    | "diff(" <set> "," <set> ")"
    | "powerset(" <set> ")"
```

2.8. Formulas

2.8.1. Connectors

A formula is a sequence of propositions (that can be variables) and connectors $\neg p$ (`not`), $\wedge$ (`and`), $\vee$ (`or`), $\oplus$ (`xor`), $\rightarrow$ (`=>`) or $\leftrightarrow$ (`<=>`).

```
<connectors(<T>)> ::=
    |     "not" <T>
    | <T> "and" <T>
    | <T> "or" <T>
    | <T> "xor" <T>
    | <T> "=>" <T>
    | <T> "<=>" <T>
```

2.8.2. Generalized connectors

Generalized connectors `bigand`, `bigor`, `exact`, `atmost` and `atleast` are also available for generalizing the formulas using sets. Here is the rule for these:

```
<generalized-connectors(<T>)> ::=
    | "bigand" <comma-list(<var>)> "in" <comma-list(<set>)>
```

```
                              ["when" <bool>] ":" <T> "end"
    | "bigor" <comma-list(<var>)> "in" <comma-list(<set>)>
                              ["when" <bool>] ":" <T> "end"
    | "exact(" <int> "," <set> ")"
    | "atmost(" <int> "," <set> ")"
    | "atleast(" <int> "," <set> ")"
```

Bigand and bigor When multiple variables and sets are given, the `bigand` and `bigor` operators will produce the `and`/`or` sequence for each possible couple of value of each set (the set of couples is the cartesian product of the given sets). For example,

The formula	expands to...
$\bigwedge\limits_{\substack{i\in[1..2]\\ j\in[a,b]}} p_{i,j}$	$p_{1,a} \wedge p_{1,b} \wedge p_{2,a} \wedge p_{2,b}$

`bigand $i,$j in [1..2],[a,b]:`
`p(i,j)`
`end`
	`p(1,a) and p(1,b)`
	`and p(2,a) and p(2,b)`

The `when` is optional and allows to apply a condition to each couple of valued variable.

On the following two examples, the math expression is given on the left and the matching touist code is given on the right:

$$\bigwedge_{\substack{i\in[1..n]\\ j\in[a,b,c]}} p_{i,j}$$

```
bigand $i,$j in [1..$n],[a,b,c]:
    p($i,$j)
end
```

$$\bigvee_{\substack{v\in[A,B,C]\\ x\in[1..9]\\ y\in[3..4]\\ x\neq y\\ x\neq A}} v_{x,y}$$

```
bigor $v,$x,$y
      in [A,B,C],[1..9],[3..4]
      when $v!=A and $x!=$y:
    $v($x)
end
```

Exact, atmost and atleast The operator `exact(3,[a,b,c,d,e])` will produce the formula that ensures that, for any valuation, exactly 3 propositions can be `true` simultanously. The operator `atleast` ensures that at least N propositions are `true` simultanously, and `atmost` does the opposite.

Note. (TODO) explain the cases N=0 or empty set

2.8.3. Simple formulas

The constants ⊤ (`Top`) and ⊥ (`Bot`) allows to express the "always true" and "always false". Here is the complete grammar:

```
<formula-simple> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <formula(<formula-simple>)>

<formula(<T>)> ::=
    | "(" <T> ")"
    | "if" <bool> "then" <T> "else" <T> "end"
    | <connectors(<T>)>
    | <generalized-connectors(<T>)>
    | <let-affect(<T>)>
```

2.8.4. SMT formulas

Touist can also be given SMT formulas and output the SMT2-compliant file. (TODO)

2.8.5. Local variables

Sometimes, you want to use the same result in multiple places. You might not be able to use a global affectation (presented in 2.2) because you are in a nested formula. The `let` construct lets you create temporary variables inside formulas:

```
<let-affect<T>> ::=
    | "let" <var> "=" <int>|<float>|<bool>|<prop>|<set> ":" <formula<T>>
```

Example:

```
;; This piece of code has no actual meaning
$letters = [a,b,c,d,e]
bigand $letter,$number in $letters,[1..card($letters)]:
  has($letter,$number) =>
  let $without_letter = diff($letters,$letter): ;; keep temorary result
  bigand $l1 in $without_letter:
      p($letter)
  end
end
```

2.9. Formal grammar

This section presents the grammar formatted in a BNF-like way. Some rules (a rule begins with "`::=`") are parametrized so that some parts of the grammar are "factorized" (the idea of parametrized rules come from the Menhir parser generator used for generating the touist parser).

Note. This grammar specification is not LL(1) and could not be implemented as such using Menhir; most of the type checking is made after the abstract syntaxic tree is produced. The only purpose of the present specification is to give a clear view of what is possible and not possible with this language.

INT = [0-9]+
FLOAT = [0-9]+\.[0-9]+
TERM = [_0-9]*[a-zA-Z][a-zA-Z_0-9]*

```
<touist-file> ::= <affect> <touist-file>
                | <formula> <touist-file>
                | EOF

<var> ::= "$" TERM
    | "$" TERM "(" <comma-list(<float> | <int> | <prop>)> ")"

<prop> ::=
    | <var>
    | TERM
    | TERM "(" <comma-list(<float> | <int> | <prop>)> ")"

<affect> ::= <var> "=" (<int>|<float>|<bool>|<prop>|<set>)

<let-affect<T>> ::=
    | "let" <var> "=" <int>|<float>|<bool>|<prop>|<set> ":" <formula<T>>

<equality(<T>)> ::=
    | <T> "!=" <T>
    | <T> "==" <T>

<order(<T>)> ::=
    | <T> ">" <T>
    | <T> "<" <T>
    | <T> "<=" <T>
    | <T> ">=" <T>

<bool> ::= "(" <bool> ")"
    | <var>
    | "true"
    | "false"
    | (<int>|<float>|<prop>|<bool>) "in" <set>
```

```
    | "subset(" <set> "," <set> ")"
    | "empty(" <set> ")"
    | <equality(<int>|<float>|<prop>)>
    | <order(<int>|<float>)>
    | <connectors(<bool>)>

<num-operation(<T>)> ::=
    | <T> "+" <T>
    | <T> "-" <T>
    |     "-" <T>
    | <T> "*" <T>
    | <T> "/" <T>

<num-operation-others(<T>)> ::=
    | <T> "mod" <T>
    | "abs(" <T> ")"

<int> ::=
    | "(" <int> ")"
    | <var>
    | INT
    | num-operation(<int>)
    | num-operation-others(<int>)
    | "if" <bool> "then" <int> "else" <int> "end"
    | "int(" (<int>|<float>) ")"
    | "card(" <set> ")"

<float> ::=
    | "(" <float> ")"
    | <var>
    | FLOAT
    | num-operation(<float>)
    | num-operation-others(<float>)
    | "if" <bool> "then" <float> "else" <float> "end"
    | "float(" (<int>|<float>)  ")"
    | "sqrt(" <float> ")"

<set> ::= "(" <set> ")"
    | <var>
    | "[" <comma-list(<int>|<float>|<prop>|<bool>)> "]"
    | "[ <int> ".." <int> "]"
    | "union(" <set> "," <set> ")"
    | "inter(" <set> "," <set> ")"
    | "diff(" <set> "," <set> ")"
    | "powerset(" <set> ")"
```

```
<comma-list(<T>)> ::= <T> | <T> "," <comma-list(<T>)>

<generalized-connectors(<T>)> ::=
    | "bigand" <comma-list(<var>)> "in" <comma-list(<set>)>
                                ["when" <bool>] ":" <T> "end"
    | "bigor" <comma-list(<var>)> "in" <comma-list(<set>)>
                                ["when" <bool>] ":" <T> "end"
    | "exact(" <int> "," <set> ")"
    | "atmost(" <int> "," <set> ")"
    | "atleast(" <int> "," <set> ")"

<connectors(<T>)> ::=
    |     "not" <T>
    | <T> "and" <T>
    | <T> "or" <T>
    | <T> "xor" <T>
    | <T> "=>" <T>
    | <T> "<=>" <T>

<formula(<T>)> ::=
    | "(" <T> ")"
    | "if" <bool> "then" <T> "else" <T> "end"
    | <connectors(<T>)>
    | <generalized-connectors(<T>)>
    | <let-affect(<T>)>

<formula-simple> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <formula(<formula-simple>)>

<formula-smt> ::=
    | <formula(<formula-smt>)>
    | <expr-smt>

<expr-smt> ::=
    | "Top"
    | "Bot"
    | <prop>
    | <var>
    | <int>
    | <float>
    | <order>(<expr-smt>)
    | <num-operations_standard(<expr-smt>)>
```

```
  | <equality(<expr-smt>)>
  | <in_parenthesis(<expr-smt>)>
```

References

[1] Khaled Skander Ben Slimane, Alexis Comte, Olivier Gasquet, Abdelwahab Heba, Olivier Lezaud, Frédéric Maris, and Maël Valais. "La Logique Facile Avec TouIST (formalisez et Résolvez Facilement Des Problèmes Du Monde Réel)." In *Actes Des 9es Journées d'Intelligence Artificielle Fondamentale (IAF 2015)*. 2015. http://pfia2015.inria.fr/actes/download.php?conf=IAF&file=Ben_Slimane_IAF_2015.pdf.

[2] Khaled Skander Ben Slimane, Alexis Comte, Olivier Gasquet, Abdelwahab Heba, Olivier Lezaud, Frederic Maris, and Mael Valais. "Twist Your Logic with TouIST." *CoRR* abs/1507.03663. 2015. http://arxiv.org/abs/1507.03663.