Touist reference manual

2017-05-09 (touist v3.0.1-26-gc9fb6a2)

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. 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 (see 3.2)
1. Type `p and q`	1. Create a file `p and q`
2. Press "Solve"	2. Type `./touist --solve yourfile`
3. Press "Next" to see other models	3. The first model is displayed

1.1. Check logical consequence

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* (or *entails*) symbol ($\models$) does not belong to the `touist` language (we call it "metalanguage"). 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

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

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. Variables

First, we describe what a variable is. Then, we detail how to affect variables (with global or local affectations).

2.2.1. Syntax of a variable

```
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
<var> ::= "$" TERM                              <- simple-var
    | "$" TERM "(" <comma-list(<expr>)> ")"     <- 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 indexes in braces, e.g., `$var($i,a,4)`. The leading variable (`$var`) must always contain a proposition. The nested indexes (e.g., `$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 2.6.3). If one of the indexes 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`	

2.2.2. Global affectation

We call “global variables” any variable that is affected in the top-level formula (meaning that global variables cannot be nested in other formulas) using the following affectation syntax:

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

Global variables apply for the whole code, even if the affectation is located before it is first used. This is because all global affectations are evaluated before any formula.

The only case where the order of affectation is important is when you want to use a variable in a global affectation expression. Global affectations are sequentially evaluated, so the order of affectation matters. For example:

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

2.2.3. Local affectation (`let` construct)

Sometimes, you want to use the same result in multiple places and you cannot use a global variable (presented in 2.2.2) because of nested formulas. The `let` construct lets you create temporary variables inside formulas:

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

The `let` affectation can only be used in formulas (detailed in 2.7) and cannot be used in expressions (`<expr>`, i.e., integer, floating-point, boolean or set expressions).

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
```

Note. The scope of a variable affected using `let` is limited to the formula that follows the colon (`:`). If this formula is followed by a whitespace and an other formula, the second formula will not be in the variable scope. Example:

```
  let $v=10: prop($v)
  prop($v)    ;; error: $v is not in scope anymore
```

2.3. Propositions

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

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 indexes of type integer, float, boolean or set.

2.3.1. Tuple proposition containing a set

A tuple proposition that is in an expression and that contains at least one set in its indexes will be expanded to a set of the cartesian product of the set indexes. This feature is called **set-building** and is described in 2.6.3 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)]`

2.4. 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.

2.4.1. 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> ")"
```

2.4.2. 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> ")"
```

2.5. 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 2.6.

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>
```

2.6. 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:

2.6.1. 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 `[]`.

2.6.2. 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 (respectively 1.0). It is not possible to change the step for now.

2.6.3. Set-builder notation

$\{p(x_1, ..., x_n) \mid (x_1, ..., x_n) \in S_1 \times ... \times S_n\}$, which the set of tuple propositions based on the cartesian product of the sets $S_1, ..., S_n$, can be written `p($S1,$S2,$S3)`. The example `p([a,b,c])` will produce `[p(a),p(b),p(c)]`. You can mix sets, integers, floats, propositions and booleans in indexes:

Proposition	produces the set
`f(1,[a,b],[7..8])`	`[f(1,a,7),f(1,a,8),` `f(1,b,7),f(1,b,8)]`

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).

2.6.4. Operators using sets

Some common set operators are available. Let P and Q denote two sets:

Type	Syntax	Math notation	Description
`<set>`	`inter($P,$Q)`	$P \cap Q$	intersection
`<set>`	`union($P,$Q)`	$P \cup Q$	union
`<set>`	`diff($P,$Q)`	$P \setminus Q$	difference
`<set>`	`powerset($Q)`	$\mathcal{P}(Q)$	powerset
`<int>`	`card($S)`	$\|S\|$	cardinal
`<bool>`	`empty($P)`	$P = \emptyset$	set is empty
`<bool>`	`$e in $P`	$e \in P$	belongs to
`<bool>`	`subset($P,$Q)`	$P \subseteq Q$	is a subset or equal

The three last operators of type `<bool>` (`empty`, `in` and `subset`) have also been described in the boolean section (2.5).

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> "]"      <- step is 1
    | "[ <float> ".." <float> "]"  <- step is 1.0
    | "union(" <set> "," <set> ")"
    | "inter(" <set> "," <set> ")"
    | "diff(" <set> "," <set> ")"
    | "powerset(" <set> ")"
```

2.7. Formulas

2.7.1. 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>
```

2.7.2. 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...
$\bigwedge\limits_{\substack{i\in\{1,\ldots,2\}\\ j\in\{a,b\}}} p_{i,j}$	$p_{1,a}\wedge p_{1,b}\wedge p_{2,a}\wedge p_{2,b}$

`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
```

Special cases for quantifier elimination Here is the list of "limit" cases where `bigand` and `bigor` will produce special results:

- In `bigand`, if a set is empty then `Top` is produced
- In `bigand`, if the `when` condition is always false then `Top` is produced
- In `bigor`, if a set is empty then `Bot` is produced
- In `bigor`, if the `when` condition is always false then `Bot` is produced

These behaviors come from the idea of quantification behind the `bigand` and `bigor` operators:

Universal quantification	$\forall x\in S, p(x)$	`bigand $x in $S: p($x) end`
Existential quantification	$\exists x\in S, p(x)$	`bigor $x in $S: p($x) end`

The following properties on quantifiers hold:

$$\begin{aligned}\forall x\in\emptyset, p(x) &\equiv \top\\ \exists x\in\emptyset, p(x) &\equiv \bot\end{aligned}\tag{1}$$

which helps understand why `Top` and `Bot` are produced.
Todo. Clarify this explanation.

Exact, atmost and atleast Touist provides some specialized operators, namely `exact`, `atmost` and `atleast`. In some cases, these operators can drastically lower the size of some formulas. The syntax of these constructs is:

Math notation	Touist syntax
$\leqslant^k_{x \in P}\ x$	`atmost($k,$P)`
$\geqslant^k_{x \in P}\ x$	`atleast($k,$P)`
$<>^k_{x \in P} x$	`exact($k,$P)`

Let P be a set of propositions, x a proposition and k a positive integer. Then:

- $\leqslant^k_{x \in P}\ x$ represents ”at any time, at most k propositions $x \in P$ must be true”
- $\geqslant^k_{x \in P}\ x$ represents ”at any time, at least k propositions $x \in P$ must be true”
- $<>^k_{x \in P} x$ represents ”at any time, exactly k propositions $x \in P$ must be true”

These operators are extremely expensive in the sense that they produce formulas with an exponential size. For example, `exact(5,p([1..20])` will produce a disjunction of $\binom{20}{5} = 15504$ conjunctions.
Note. The notation `p([1..20])` is called “set-builder” and is defined in 2.6.3. Using this syntax, the formula `exact(5,p([1..20])` is equivalent to

$$<>^k_{x \in P} p(x)$$

Special cases for quantifier elimination The following table sums up the various “limit” cases that may not be obvious. In this table, k is a positive integer and P is a set of propositions.

	`$k`	`$P`	**Gives**
`exact($k,$P)`	$k = 0$	$P = \emptyset$	`Top`
	$k = 0$	$P \neq \emptyset$	`bigand $p in $P: not $p end`
	$k > 0$	$\lvert P\rvert = k$	`bigand $p in $P: $p end`
`atleast($k,$P)`	$k = 0$	any	`Top`
	$k = 1$	any	`bigor $p in $P: $p end`
	$k > 0$	$\emptyset$	`Bot` (subcase of next row)
	$k > 0$	$\lvert P\rvert < k$	`Bot`
	$k > 0$	$\lvert P\rvert = k$	`bigand $p in $P: $p end`
`atmost($k,$P)`	$k = 0$	$\emptyset$	`Top` (subcase of next row)
	$k = 0$	any	`Top`

How to read the table: for example, the row where $k > 0$ and $|P| < k$ should be read ”when using `atleast`, all couples $(k, P) \in \{(k, P)|k > 0, |P| < k\}$ will produce the special case `Top`”.

2.7.3. Propositional logic formulas

The constants $\top$ (`Top`) and $\bot$ (`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>)>
```

2.7.4. SMT formulas

Touist can also be given Satisfiability Modulo Theory (SMT) formulas and output the SMT2-LIB-compliant file.

Todo. Describe the SMT language

2.8. 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 LR(1) and could not be implemented as such using Menhir; most of the type checking is made after the abstract syntactic 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

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

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

<affect> ::= <var> "=" (<expr>)

<let-affect<T>> ::=
    | "let" <var> "=" <expr> ":" <formula<T>>

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

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

<bool> ::= "(" <bool> ")"
    | <var>
    | "true"
    | "false"
    | (<expr>) "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(<expr>)> "]"
    | "[ <int> ".." <int> "]"      <- step is 1
    | "[ <float> ".." <float> "]"  <- step is 1.0
    | "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>)>
```

3. Command-line tool (`touist`)

3.1. Installation

The main tool that parses and solves the touist programs is written in Ocaml. It is easily installable (as long as you have installed `ocaml` and `opam`) with the command

```
opam install touist
```

Note. `touist` can only solve SAT problems (written using propositional logic). Problems written using the Satisfiability Modulo Theory (SMT) extension of `touist` cannot (yet) be solved, but can still be translated to the SMT2-LIB format which can then be fed to a SMT solver.

3.2. Usage

Any touist command is of the form:

```
touist [-o OUTPUT] (INPUT | -) [options...]
```

The flags can be given in any order. You can use the standard input (`stdin`) instead of using an input file by setting the `-` argument. With no `-o` flag, touist will output to the standard output (`stdout`).

3.2.1. Usage for propositional logic (SAT mode)

The language accepted for propositional logic is described in 2.7.3. This mode is enabled by default, but can be optionally expressed using the `--sat` flag.

With no other argument, touist will simply translate the touist code to the DIMACS format and then output the mapping table (that maps each proposition to an integer > 0) in DIMACS comments. You can redirect this mapping table using the `--table <filename>` flag.

Options: `--solve`. Ask touist to solve the SAT problem. By default, the first model is displayed; you can ask for more models using the `--limit N` option. The models are separated by lines beginning with `====` and for one model, each line contains a valuation followed by the corresponding proposition. For example:

```
echo a and b | touist - --solve
```

will display

```
==== model 0
1 b
1 a
==== Found 1 models, limit is 1 (--limit N for more models)
```

which corresponds to the valuation $\{a \leftarrow 1, b \leftarrow 1\}$. Note that the model counter begins at 0. With this format, you can easily filter the results. For example, the following command will only show the propositions that are `true`:

```
echo a and b | touist - --solve | grep ^1
```

`--limit N`. In conjunction with `--solve`, set the maximum number of models returned. When N=0, all models are returned.

`--count`. Instead of returning the models, just return the count of models. This option will only work for small problems: the number of models explodes when the number of propositions is big.

3.2.2. Other options

`--latex`. Translates the given touist code to LaTeX. The resulting latex code only require the `mathtools` package and is compatible with Mathjax (JavaScript tool for displaying math in HTML). This command does not print `\begin{document}` nor any latex headers (`\usepackage{}`...).

`--show`. This option prints the formula generated by the given touist file. This is useful for debugging and testing that the constructs `bigand`, `bigor`, `exact`... are correclty evaluated.

`--show-hidden`. This is specific to the SAT mode. When displaying the DIMACS result, also include the hidden propositions that have been generated during the CNF expansion by the Tseitin transformation.

`--linter`. This option disables all outputs except for errors. It also shortens then evaluation step by bypassing the expansive `bigand`, `exact`, `powerset`... constructs.

`--detailed-position`. Adds the absolute character position in error and warning messages with for format `line:col:abs_first:abs_last: message`.

`--debug-syntax`. This is a development option that adds to the error and warning messages the state number of the LL(1) automaton. Each state number that may trigger a syntax error should have a corresponding message in `src/parser.messages`.

`--debug-cnf`. This is also a development option; in SAT mode, it prints the successive recursive transformations that produce the CNF formula.

3.2.3. Usage for Satisfiability Modulo Theory (SMT mode)

The language accepted by this mode is described in 2.7.4. The flag `--smt LOGIC` enables the SMT mode.
Todo. Explain which `LOGIC` can be given and how to use `--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.