Touist reference manual

2017-07-17 (touist v3.4.0)

Olivier Gasquet

Frédéric Maris

Maël Valais

Chapter 1

Introduction

1.1. What is TouIST?

TOUIST[1] is a language that allows us 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`, a command-line program written in OCaml that compiles and solves TOUIST problems (see 3 for usage).

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

1.2. 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.

---

[1] **Tou**louse **I**ntegrated **S**atisfiability **T**ool. It is prononced *twist*. We were looking for a memorable and pronounceable name that had no homonym on Google. And it had to sound like fun, too!

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
```

1.3. Contribute to this document

The source of this document is written in madoko, a markdown dialect. This language has been chosen over Latex for its brevity (Pandoc could have also been chosen). You can contribute to this document through a pull-request or by discussing your thoughts in the issue tracker. To edit it, you can first take a look at what the madoko editor can do (with this link, the images and colorizer will not work), or see how to edit properly in `CONTRIBUTE-TO-DOCS.md` (located in `docs/`).
Note. This reference manual still lacks of a step-by-step tutorial as well as illustrating examples. Feel free to contribute!

Chapter 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[2] list of affectations and formulas. Affectations are global and can be interlaced with formulas as long as they are not nested in formulas (for local variables, you can use `let`, see 2.2.3). Comments begin with the ";;" sequence. Two backslashes (\\) in a formula will produce a new line in the latex view.

Note. The whitespace-separated list of formulas is actually going to be converted to a conjunction; it avoids many intermediate **and**. **Warning:** each formula in this list is going to be put into parenthesis:

```
a or b
c => a
```

will be translated to

```
(a and b) and (c => a)
```

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.

---

[2]A whitespace is a space, tab or newline.

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
    | "let" <comma-list(<var>)> "=" <comma-list(<expr>)> ":" <formula<T>>
<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 purpose
$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
```

You can also chain multiple variables in a single `let`:

```
    let $E,$x,$y = [1..2],3,4: ...
```

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). Parenthesis can be used. The order of priority for the infix operators is:

highest priority	mod
	*,/
lowest priority	+,-

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 explicitly 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.

Parenthesis can be used in boolean expressions. The priority order for booleans is:

highest priority	`==,!=,<=,>=,<,>, in`
	`not`
	`xor`
	`and`
	`or`
lowest priority	`=>, <=>`

Note that `=>` and `<=>` associativity is evaluated from right to left.

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>
```

Note. You can chain multiple variables in `bigand` or `bigor` by giving a list of variables and sets. This will translate into nested `bigand`/`bigor`. You can even use the value of outer variables in inner set declarations:

```
bigand $i,$j in [1..3], [1..$i]:
    p($i,$j)
end
```

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 us 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:

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

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

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

```
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

The TOUIST language 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
$\leqslant_{x \in P}^{k} x$	`atmost($k,$P)`
$\geqslant_{x \in P}^{k} x$	`atleast($k,$P)`
$<>_{x \in P}^{k} x$	`exact($k,$P)`

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

- $\leqslant_{x \in P}^{k} x$ represents "at any time, at most k propositions $x \in P$ must be true"
- $\geqslant_{x \in P}^{k} x$ represents "at any time, at least k propositions $x \in P$ must be true"
- $<>_{x \in P}^{k} x$ 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

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

Example 1.
`exact(2,[a,b,c])` is equivalent to

$$(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$$

Special cases for operator 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`
	$k > 0$	$|P| = k$	`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)
	$k > 0$	$|P| < k$	`Bot`
	$k > 0$	$|P| = k$	`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`) allow us 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. SMT formulas

The TOUIST language also accepts formulas of the Satisfiability Modulo Theory (SMT). To use the `touist` program with SMT formulas, see 3.2.4
Todo. Describe the SMT language

2.9. QBF formulas

The TOUIST language accepts Quantified Boolean Formulas (QBF). Using the `--qbf` and `--solve` flags (in `touist`) or the QBF selector (in the graphical interface), you can solve QBF problems with existential and universal quantifiers over boolean values. This logic is basically the same as the SAT grammar, except for two new operators $\exists$ (`exists`) and $\forall$ (`forall`):

```
<formula-qbf> ::=
    [...]
    | "exists" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>
    | "forall" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>

<for> ::= "for" <var> "in" <set>
```

One quantifier can quantify over multiple propositions. For example:

```
    forall e,d: (exists a,b: a => b) => (e and forall c: e => c)
```

$$\forall e.\forall d.(\exists a.\exists b.(a \Rightarrow b)) \Rightarrow (e \wedge \forall c.(e \Rightarrow c))$$

2.9.1. Free variables

Any free variable (i.e., not quantified) will be existentially quantified in an inner-most way. For example, $\forall a.a \wedge b$ will become $\forall a.\forall b.(a \wedge b)$.

2.9.2. Renaming during the prenex transformation

Using the rules for transforming an arbitrary formula into prenex form requires sometimes some renaming. For example, $(\forall a.a) \wedge a$ must be transformed into $\forall a_1.(a_1 \wedge a))$.

Sometimes, the renaming is not possible. For example, in $\forall a.(\exists a.a)$, we cannot guess which quantifier the rightmost a should be bound to. In this case, `touist` will give and error.

Also, the use of the whitespace-as-formula-separator can lead to some misunderstanding. For example, in formula

```
    exists x: x
    x => y
```

the `x` in the second line is free whereas the `x` in the first line is bounded. This is because two whitespace-separated formulas have different scopes. In fact, the previous formula could be re-written as

```
    (exists x: x) and (x => y)
```

The only way to span the scope of a bounded variable across multiple lines is to continue the formula with a `and` instead of using a new line:

```
exists x: x
and (x => y)
```

2.9.3. Multiple `exists` or `forall`

The keyword `for` used in `exists` or `forall` allow us to generate multiple quantifiers using sets. Imagine that you want to write

$$\exists a.\exists b.\exists c.\exists d.\Phi$$

You can use one of the two notations:

```
exists a: exists b: exists c: exists d: phi ;; (1)
exists $p for $p in [a,b,c,d]: phi              ;; equivalent to (1)
```

2.10. 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>>
    | "let" <comma-list(<var>)> "=" <comma-list(<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> | <T> "\\"            <- newline marker for latex display
    | "(" <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>)>

<formula-qbf> ::=
    | "Top"
    | "Bot"
```

```
    | <prop>
    | <var>
    | <formula(<formula-qbf>)>
    | "exists" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>
    | "forall" <comma-list(<prop>|<var>)> [<for>] ":" <formula-qbf>

<for> ::= "for" <var> "in" <set>
```

Chapter 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. By default, `touist` only comes with a SAT solver. Problems written in TOUIST using the Satisfiability Modulo Theory (SMT) or Quantified Boolean Formulas (QBF) grammars can also be used (see 3.2.4 and 3.2.5).
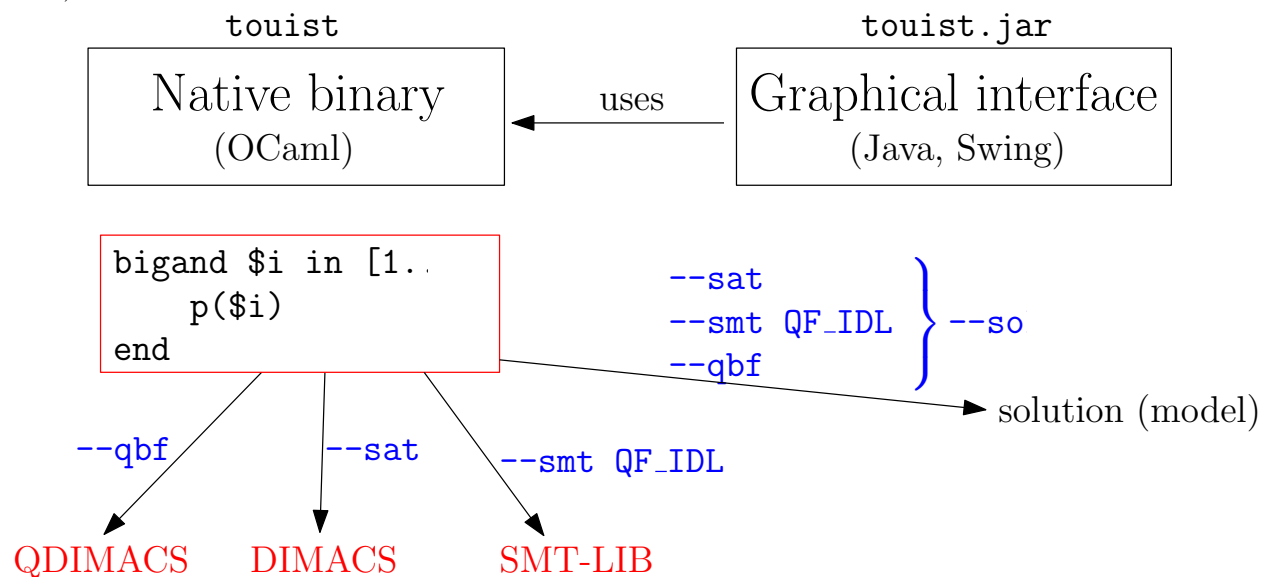
3.2. Usage

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. Return codes

Code	Label	Description
0	OK	
1	UNKNOWN	
2	CMD_USAGE	something wrong with the command-line arguments
3	CMD_UNSUPPORTED	
4	TOUIST_SYNTAX	any error of syntax, type or semantic
5	TOUIST_TIMEOUT	
6	TOUIST_MEMORY	
7	TOUIST_UNKNOWN	
8	SOLVER_UNSAT	the solver did not find any model
9	SOLVER_UNKNOWN	
10	SOLVER_TIMEOUT	
11	SOLVER_MEMORY	

Note that the `SOLVER_UNSAT` is not really an error, but we choose to return a non-zero return code anyway.

3.2.2. 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.

SAT solver

By default, `touist` embeds Minisat (see [5]), a SAT solver written in C++ at the Chalmers University of Technology, Sweden. It is distributed under the MIT license. To be able to embed it into OCaml, we use the binding ocaml-minisat which relies on a C version of Minisat (Minisat-C-1.14.1) for portability concerns.

`--solve` asks `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 return a single model (and in this case, there is only one single model):

```
1 a
1 b
```

Each line corresponds to a valuation, and each valuation should be read `value proposition`. In the example, a takes the value 1 (true). 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
```

`--solve --limit N` allows to display multiple models. With this 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, with `--limit 0` which displays an unlimited number of models:

```
echo a or b | touist - --solve --limit 0
```

will display

```
==== model 0
1 a
0 b
==== model 1
0 a
```

```
1 b
==== model 2
1 a
1 b
==== found 3 models, limit is 0 (--limit N for more models)
```

Note that the model counter begins at 0.

--solve --count tries to return the count of models instead of returning the models. This option will only work for small problems: the number of models explodes when the number of propositions is big.

3.2.3. Other general options

The following general options must be using in conjunction of **--smt** , **--qbf** or **--sat** (**--sat** is used by default and can be omitted).

--latex, **--latex-full** translate the given TOUIST code to LATEX. Two options are available:

- with **--latex**, only the math formulas are translated and no header/footer is added (e.g., `\begin{document}`). This mode is compatible with any *light* latex processors (e.g., `mathjax` or `jlatexmath`).
- with **--latex-full**, a proper header is inserted so that the output can directly be given to `pdflatex` or any other fully-featured latex processor. The `mathtools` package is necessary for `\begin{pmatrix*}` when matching parenthesis span across multiple lines.

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

--error-format allows the user to format the errors as you wish. This argument can be useful for plugging `touist` to another program that needs to read the errors. The placeholders you can use are:

%f	file name
%l	line number (start)
%L	line number (end)
%c	column number (start)
%C	column number (end)
%b	buffer offset (start)
%B	buffer offset (end)
%t	error type (*warning* or *error*)
%m	error message
\n	new line

By default, the errors and warnings will display with the formatting `%f: line %l, col %c-%C: %t: %m`. An ending newline (`\n`) is automatically added.

--wrap-width N lets you choose the width of the messages (errors, warnings) given by `touist`. By default, `touist` will wrap all messages with a 76 characters limit. With N set to 0, you can disable the wrapping.

--debug-syntax 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/lib/parser.messages`.

--debug-cnf (only with **--sat** or **--qbf**)) is also a development option; in SAT and QBF modes, it prints the successive recursive transformations that produce the CNF formula.

3.2.4. Usage for Satisfiability Modulo Theory (SMT mode)

The language accepted by the SMT mode is described in 2.8.

By default, `touist` is able to translate problems into the SMT-LIB format with the `--smt LOGIC` flag. In this mode, `LOGIC` can be any non-whitespace string of characters (which will be transformed in uppercase automatically). `LOGIC` will simply be used to fill the correct field in the `SMT-LIB` file.

SMT solver

TOUIST can embed an optional SMT solver, Yices 2.5.2 (see [4]). It is developed by SRI (Stanford Research Institute, California) and is written in C. It is free to use for non-commercial purposes. Its code is licensed under a restrictive non-commercial EULA which the user must agree before using (see the license). To enable it, you need to install the OCaml binding `yices2` which embeds the Yices sources:

```
opam install yices2
```

If `touist` was previously installed, it will be re-installed to support the newly installed `yices2`.

When using both `--smt LOGIC` and `--solve`, the `LOGIC` argument must be one of the logics Yices 2.5.2 supports. Here is a table with the logics that are can be used through the TOUIST language:

LOGIC	Meaning
QF_IDL	Integer Difference Logic
QF_RDL	Real Difference Logic
QF_LIA	Linear Integer Arithmetic
QF_LRA	Linear Real Arithmetic

Note that *QF* means quantifier-free. Also note that you can solve any SAT problem using any SMT logic solver.

For example:

```
echo 'x > 3' | touist --smt QF_IDL --solve -
```

which will give you the model

```
4 x
```

which should be read as x takes the value 4.

3.2.5. Usage for quantified boolean logic (QBF mode)

For now, the `QDIMACS` format (which is the equivalent of `DIMACS` for quantified boolean formulas) cannot be printed from a TOUIST file. You can still solve problems using both `--qbf` and `--solve` (see below).

QBF solver

`touist` can embed an optional QBF solver, Quantor 3.2 (see [3]). It is developed at the Johannes Kepler University, Austria. Its source is under a (non-restrictive) BSD license. To enable the QBF solver, you must install the OCaml binding `qbf` which embeds the Quantor source:

```
opam install qbf
```

Then, we can solve a small example:

```
echo 'forall x: x or (exists y: y)' | touist --qbf --solve -
```

which will give you a partial model:

```
? x
? y
```

where `?` means that this value is undefined. To get an actual mode, we must *force* the valuations (and doing so, we explore the tree of possible valuations). `touist` and its graphical interface are unable to force these valuations and explore the tree. As a consequence, they are also unable to visualize the tree (where each leaf would be a model). For now, the only way to do it is by hand.

Chapter 4

Technical details

4.1. One single syntax error per run

You might have noticed that on every run of `touist`, only one error is shown at a time. Many compilers are able to show multiple errors across the file (e.g., any C compiler). Some other compilers, like OCaml, only display one error at a time. This feature is often expected by developers as a time saver: one single run of the compiler to squash as many mistakes as possible.

This feature is tightly liked to one particular trait of the grammar of the language: the semi-colon (;). When an error comes up, the semi-colon will act as a checkpoint to which the parser will try to skip to. Hoping that this error is only contained in this instruction, the parser will try to continue after the semi-colon.

The TOUIST grammar does not have such an instruction marker; because of that, we are not able to skip to the next instruction.

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.

[3] Armin Biere. "Resolve and Expand." In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, 59–70. SAT'04. Springer-Verlag, Berlin, Heidelberg. 2005. doi:10.1007/11527695_5.

[4] Bruno Dutertre. "Yices 2.2." In *Computer-Aided Verification (CAV'2014)*, edited by Armin Biere and Roderick Bloem, 8559:737–744. Lecture Notes in Computer Science. Springer. Jul. 2014.

[5] Niklas Sörensson, and Niklas Een. "MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization (SAT-2005)." In . 2002.