# 1. Introduction

1.1. What is TouIST? .................................................. 3
1.2. Check logical consequence .................................. 3
1.3. Contribute to this document ................................. 4

# 2. Language reference

2.1. Structure of a TouIST file .................................. 5
2.2. Variables .......................................................... 5
   2.2.1. Syntax of a variable ................................... 5
   2.2.2. Global affectation ...................................... 6
   2.2.3. Local affectation (let construct) ................. 6
2.3. Propositions ..................................................... 7
   2.3.1. Tuple proposition containing a set ............... 7
2.4. Numeric expression ............................................ 7
   2.4.1. Integers .................................................. 8
   2.4.2. Floats .................................................... 8
2.5. Booleans .......................................................... 8
2.6. Sets ............................................................... 9
   2.6.1. Sets defined by enumeration ....................... 9
   2.6.2. Sets defined by a range .......................... 9
   2.6.3. Set-builder notation (list comprehension) .... 9
   2.6.4. Operators using sets .............................. 10
2.7. Formulas .......................................................... 11
   2.7.1. Connectors .............................................. 11
   2.7.2. Newline-and ........................................... 11
   2.7.3. Generalized connectors ......................... 12
   2.7.4. Propositional logic formulas ................. 13
   2.7.5. SMT formulas ......................................... 14
   2.7.6. QBF formulas ......................................... 14
   2.7.7. Quoted formulas (formulas as expressions) .... 15
2.8. Formal grammar .................................................. 15

# 3. Command-line tool (touist)

3.1. Installation ..................................................... 19
3.2. Usage ............................................................ 19
   3.2.1. Exit status ......................................... 20
   3.2.2. Usage for propositional logic (SAT mode) .... 20
   3.2.3. Other general options .......................... 21
   3.2.4. Usage for Satisfiability Modulo Theory (SMT mode) .............................. 22
   3.2.5. Usage for quantified boolean logic (QBF mode) .............................. 23
3.3. Using external solvers from touist (--solver) ........ 23

# 4. Technical details

4.1. One single syntax error per run ......................... 25

References .......................................................... 27
Chapter 1

Introduction

1.1. What is TouIST?

TouIST\(^1\) is a language that allows us to express propositional logic [1, 2, 5]. 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).

<table>
<thead>
<tr>
<th>Propositional logic</th>
<th>TouIST language</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg p)</td>
<td>not p</td>
</tr>
<tr>
<td>(p \land q)</td>
<td>p and q</td>
</tr>
<tr>
<td>(p \lor q)</td>
<td>p or q</td>
</tr>
<tr>
<td>(p \oplus q)</td>
<td>p xor q</td>
</tr>
<tr>
<td>(p \rightarrow q)</td>
<td>p =&gt; q</td>
</tr>
<tr>
<td>(p \iff q)</td>
<td>p &lt;=&gt; q</td>
</tr>
</tbody>
</table>

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 \textit{satisfiable}. For example, a model of \(p \lor q\) is \(\{p = \text{true}, q = \text{false}\}\). To check the models of this formula using TouIST, you can do

<table>
<thead>
<tr>
<th>Graphical Java interface</th>
<th>Command-line interface (see 3.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Type (p) and (q)</td>
<td>1. Create a file (p) and (q)</td>
</tr>
<tr>
<td>2. Press “Solve”</td>
<td>2. Type touist --solve yourfile</td>
</tr>
<tr>
<td>3. Press “Next” to see other models</td>
<td>3. The first model is displayed</td>
</tr>
</tbody>
</table>

1.2. Check logical consequence

From a wikipedia example:

\begin{tabular}{|l|}
\hline
\textbf{Premise 1:} & If it’s raining then it’s cloudy \\
\textbf{Premise 2:} & It’s raining \\
\textbf{Conclusion:} & It’s cloudy \\
\hline
\end{tabular}

This inference can be written

\[ \{\text{raining} \rightarrow \text{cloudy}, \text{raining}\} \models \text{cloudy} \]

The \textit{infer} (or \textit{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.

\textbf{Theorem 1.}

Let \(H\) be a set of formulas (called \textit{hypotheses} or \textit{premises}) and \(C\) a formula (called \textit{conclusion}). Then \(H \models C\) if and only if \(H \cup \{\neg C\}\) is unsatisfiable.

\(^1\text{Toulouse Integrated Satisfiability Tool. It is pronounced twist. We were looking for a memorable and pronounceable name that had no homonym on Google. And it had to sound like fun, too!}\)
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 \rightarrow 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 \rightarrow cloudy) \text{ and } raining \text{ and } not \text{ 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 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 \texttt{let}, see 2.2.3). Comments begin with the ";;" sequence. Two backslashes (\\) in a formula will produce a new line in the latex view.

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

\begin{itemize}
  \item a or b
  \item c \Rightarrow a
\end{itemize}

will be translated to

\begin{itemize}
  \item (a or b) and (c \Rightarrow a)
\end{itemize}

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>
   | ""<formula-simple>"" <- quoted formula, since TouIST >= 3.5.1
<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 or a quoted formula. 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 \textit{predicate})

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

\footnote{A whitespace is a space, tab or newline.}
A tuple variable will always be expanded to a proposition or a quoted formula. For example, if $\text{var}=p$ 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:

<table>
<thead>
<tr>
<th>Simple-var</th>
<th>Tuple-var</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$\text{place}(\text{number})$</td>
</tr>
<tr>
<td>$\text{time}$</td>
<td>$\text{action}(i,j)$</td>
</tr>
<tr>
<td>$\text{SIZE}$</td>
<td></td>
</tr>
<tr>
<td>$\text{is_over}$</td>
<td></td>
</tr>
</tbody>
</table>

### 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:

```plaintext
<\text{affect}> ::= <\text{var}> "=" (<\text{expr}>)  \quad \text{ <-- global affectation}
<\text{expr}> ::= <\text{int}|<\text{float}|<\text{prop}|<\text{bool}|<\text{set}>
       | """<\text{formula\_simple}>"""  \quad \text{ <-- TouIST \geq 3.5.1}
```

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:

```plaintext
$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:

```plaintext
<\text{let\_affect<T>>} ::= 
| "let" <\text{var}> "=" <\text{expr}>
| "let" <\text{comma-list(<\text{var}>)}>
<\text{expr}> ::= <\text{int}|<\text{float}|<\text{prop}|<\text{bool}|<\text{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:

```plaintext
;; This piece of code has no actual purpose
$\text{letters} = [a,b,c,d,e]
\text{bigand} \;
\text{letter},\text{number} \;
\text{in} \;
\text{letters},[1..\text{card}(\text{letters})]:
\text{has}(\text{letter},\text{number}) =)
\text{let} \;
\text{without\_letter} = \text{diff}(\text{letters},\text{letter}): ;; \text{keep temorary result}
\text{bigand} \;
\text{ll} \;
\text{in} \;
\text{without\_letter}:
\;
p(\text{letter})
\quad\text{end}
\quad\text{end}
```

You can also chain multiple variables in a single let:

```plaintext
\text{let} \;
\text{E},\text{x},\text{y} = [1..2],3,4: ~
```

**Note.** The scope of a variable affected using let is limited to the formula that follows the colon (:) and if this formula is followed by a whitespace and another formula, the second formula will not be in the variable
2.3. Propositions

Scope. Example:

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

2.3. Propositions

```plaintext
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:

<table>
<thead>
<tr>
<th>Proposition</th>
<th>is in a formula</th>
<th>is in an expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>p([a])</td>
<td>p([a])</td>
<td>p(a)</td>
</tr>
<tr>
<td>p([a,b,c])</td>
<td>p([a,b,c])</td>
<td>[p(a),p(b),p(c)]</td>
</tr>
<tr>
<td>p([a,b],[1..2])</td>
<td>p([a,b],[1..2])</td>
<td>[p(a,1),p(b,1)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p(a,2),p(b,2)]</td>
</tr>
</tbody>
</table>

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:

```plaintext
<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:
• \texttt{card([a,b])} returns an integer,
• \texttt{sqrt(3)} returns a float.

\section*{2.4.1. Integers}

An integer constant \texttt{INT} is a number that satisfies the regular expression \([0-9]+\). Here is the rule for writing correct integer expressions:

\[
\texttt{<int>} ::= \begin{array}{l}
\texttt{| "(" <int> ")"} \\
\texttt{| <var>} \\
\texttt{| INT} \\
\texttt{| num-operation(<int>)} \\
\texttt{| num-operation-others(<int>)} \\
\texttt{| "if" <bool> "then" <int> "else" <int> "end"} \\
\texttt{| "int" (}\texttt{<int>|<float>}) \\
\texttt{| "card" (}\texttt{<set>})
\end{array}
\]

\section*{2.4.2. Floats}

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

\[
\texttt{<float>} ::= \begin{array}{l}
\texttt{| "(" <float> ")"} \\
\texttt{| <var>} \\
\texttt{| FLOAT} \\
\texttt{| num-operation(<float>)} \\
\texttt{| num-operation-others(<float>)} \\
\texttt{| "if" <bool> "then" <float> "else" <float> "end"} \\
\texttt{| "float" (}\texttt{<int>|<float>}) \\
\texttt{| "sqrt" (}\texttt{<float>})
\end{array}
\]

\section*{2.5. Booleans}

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

\[
\begin{array}{ll}
\texttt{\$P$ subset $Q$} & P \subseteq Q \quad P \text{ is a subset (or is included in) } Q \\
\texttt{\emptyset(P)} & P = \emptyset \quad P \text{ is an empty set} \\
\texttt{i in $P$} & i \in P \quad i \text{ is an element of the set } P
\end{array}
\]

Sets are detailed in 2.6.

\textbf{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:

\begin{tabular}{l}
\texttt{highest priority} \texttt{==,!=,<=$=$,\geq,<,\leq,\in} \\
\texttt{not} \\
\texttt{xor} \\
\texttt{and} \\
\texttt{or} \\
\texttt{lowest priority} \texttt{\Rightarrow, \Leftrightarrow}
\end{tabular}

\texttt{Note that \Rightarrow and \Leftrightarrow associativity is evaluated from right to left.}

Here is the full grammar rule for booleans:
2.6. Sets

Sets can contain anything (propositions, integers, floats, booleans, quoted formulas 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, quoted formulas or sets (or a variable of these six types). The empty set \(\emptyset\) is denoted by \(\[]\). Examples:

\[
[1,2,3+1] \\
[a,b,p(1,v)] \\
[[1,2],[3,4,5]] \\
["a or b", "c => d"]
\]

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 (list comprehension)

A set-builder expression is a set defined as \(\{p(x_1,\ldots,x_n) \mid (x_1,\ldots,x_n) \in S_1 \times \ldots \times S_n\}\), which is the set of expressions based on the cartesian product of the sets \(S_1,\ldots,S_n\). You can use the “list comprehension” (since version 3.5.2) to do that:

\[
[p($i,$j,$k) \text{ for } $i,$j,$k \text{ in } $S_1,$S_2,$S_3]
\]

List comprehension allows you to generate sets containing any expression: numbers, propositions and even formulas. In order to use formulas, you must use the quoted notation (see 2.7.7). The \texttt{when} keyword helps filter the generated elements (like in \texttt{bigand} or \texttt{bigoor}). Examples:

\[
[\$i \text{ for } \$i \text{ in } [1..100] \text{ when } \$i \text{ mod 3 == 0}];; \text{ set of integers}
[\texttt{f($i,$j)} \text{ for } \$i,$j \text{ in } [1..3],[a,b,c]] ;; \text{ set of propositions}
[\texttt{f(1,$p)} \text{ for } \$p \text{ in } [a,b,c,d] \text{ when } \$j \text{ != a}] ;; \text{ set of propositions}
["\$a \text{ and } \$b" \text{ for } \$a,$b \text{ in } [r,s],[x,y]] ;; \text{ set of quoted formulas}
\]
For simple list comprehension (without when), you can use the condensed syntax: \( f(1, [a, b], [7..8]) \) which is equivalent to \( \{ f(1, i, j) \mid i, j \in [a, b], [7..8] \} \).

<table>
<thead>
<tr>
<th>List comprehension</th>
<th>( { f(1, i, j) \mid i, j \in [a, b], [7..8] } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condensed syntax</td>
<td>( f(1, [a, b], [7..8]) )</td>
</tr>
<tr>
<td>Produced set</td>
<td>( { f(1, a, 7), f(1, a, 8), f(1, b, 7), f(1, b, 8) } )</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Math notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;set&gt;</td>
<td>$P inter $Q</td>
<td>( P \cap Q )</td>
<td>intersection</td>
</tr>
<tr>
<td>&lt;set&gt;</td>
<td>$P union $Q</td>
<td>( P \cup Q )</td>
<td>union</td>
</tr>
<tr>
<td>&lt;set&gt;</td>
<td>$P diff $Q</td>
<td>( P \setminus Q )</td>
<td>difference</td>
</tr>
<tr>
<td>&lt;set&gt;</td>
<td>powerset($Q)</td>
<td>( \mathcal{P}(Q) )</td>
<td>powerset</td>
</tr>
<tr>
<td>&lt;int&gt;</td>
<td>card($S)</td>
<td></td>
<td>cardinal</td>
</tr>
<tr>
<td>&lt;bool&gt;</td>
<td>empty($P)</td>
<td>( P = \emptyset )</td>
<td>set is empty</td>
</tr>
<tr>
<td>&lt;bool&gt;</td>
<td>$e$ in $P</td>
<td>( e \in P )</td>
<td>belongs to</td>
</tr>
<tr>
<td>&lt;bool&gt;</td>
<td>$P subset $Q</td>
<td>( P \subseteq Q )</td>
<td>is a subset or equal</td>
</tr>
</tbody>
</table>

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

The priority on operators is:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest priority</td>
<td>inter</td>
</tr>
<tr>
<td>lowest priority</td>
<td>union, diff</td>
</tr>
</tbody>
</table>

**Note.** Up to TOUIST v3.2.3, the operators inter, union, diff and subset were prefix operators (e.g., \( \text{inter}(\$A, \$B) \)). From v3.4.0 and later, these prefix operators are deprecated (but still usable). Instead, we provide more human-friendly infix operators (e.g., \( \$A \text{ inter} \$B \)).

**Powerset**

The \( \text{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: \( \text{powerset}([1, 2]) \) generates \( \emptyset, [1], [2], [1, 2] \).

Here is the complete rule for sets:

```plaintext
<set> ::= "(" <set> ")"
  | "[" <comma-list(<int>|<float>|<prop>|<bool>)> "]"
  | "[ <int> ",..", <int> "]"  \(<-\) step is 1
  | "[ <float> ",..", <float> "]"  \(<-\) step is 1.0
  | <set> "inter" <set>
  | <set> "union" <set>
  | <set> "diff" <set>
  | "powerset(" <set> ")"
```
2.7. Formulas

2.7.1. Connectors

A formula is a sequence of propositions (that can be variables) and connectors \(-p\) (not), \(\land\) (and), \(\lor\) (or), \(\oplus\) (xor), \(\rightarrow\) (\(\Rightarrow\)) or \(\leftrightarrow\) (\(\Leftrightarrow\)).

\[
<\text{connectors}(<\text{formula}<T>>) ::= \\
| "not" <T> \\
| <T> "and" <T> \\
| <T> "or" <T> \\
| <T> "xor" <T> \\
| <T> "=>" <T> \\
| <T> "<=>" <T>
\]

Parenthesis can be used in formulas in order to express priority. The default operator priority is:

\[
\begin{array}{c|c}
\text{highest priority} & not \\
& xor \\
& and \\
& or \\
& \rightarrow, \leftrightarrow \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{lowest priority} & \text{newline-and (2.7.2)} \\
\end{array}
\]

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:

\[
\text{bigand } \$1,\$j \text{ in } [1..3], [1..\$i]: \\
p(\$i,\$j)
\]

end

2.7.2. Newline-and

As mentioned in a note (first section), in top-level, a new line (or any kind of white spaces) separating two formulas will be translated into a lesser-priority and. It is expressed in the grammar as:

\[
<\text{formula}(<T>)>::= \\
| . . . \\
| <T> ("\n"|" ") <T> <- newline/whitespace in top-level is an 'and'
\]

This notation is related to the idea of a set of formulas (premises). For example, a new line would allow to express the separation of these two formulas:

\[
\{a, a \Rightarrow b, \neg c\}
\]

You can write this either with

\[
a \\
a \Rightarrow b \\
\neg c
\]

or

\[
a \ a \Rightarrow b \ not \ c
\]

that are equivalent to

\[
a \land (a \Rightarrow b) \land (\neg c)
\]

The important thing to notice is that this whitespace-and has a lower priority than any other connector.
2.7.3. Generalized connectors

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

\[
<\text{generalized-connectors}(\langle T \rangle)> ::= \\
| "\text{bigand}" \langle \text{comma-list}(\langle \text{var} \rangle)\rangle \ "in" \langle \text{comma-list}(\langle \text{set} \rangle)\rangle \\
| ["when" \langle \text{bool} \rangle : "\langle T \rangle " \text{end}]
\]

\[
| "\text{bigor}" \langle \text{comma-list}(\langle \text{var} \rangle)\rangle \ "in" \langle \text{comma-list}(\langle \text{set} \rangle)\rangle \\
| ["when" \langle \text{bool} \rangle : "\langle T \rangle " \text{end}]
\]

\[
| "\text{exact}("\langle \text{int}\rangle", "\langle \text{set}\rangle")"
\]

\[
| "\text{atmost}("\langle \text{int}\rangle", "\langle \text{set}\rangle")"
\]

\[
| "\text{atleast}("\langle \text{int}\rangle", "\langle \text{set}\rangle")"
\]

**Bigand and bigor**

When multiple variables and sets are given, the \texttt{bigand} and \texttt{bigor} operators will produce the \texttt{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,

\[
\bigland_{i,j \in [1,..2]} p_{i,j}
\]

\[
\text{bigand } \langle i,j \rangle \text{ in } [1..2],[a,b]: \\
p(i,j)
\text{end}
\]

The \texttt{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 TōIST code is given on the right:

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

\[
\text{bigand } \langle i,j \rangle \text{ in } [1..n],[a,b,c]: \\
p(i,j)
\text{end}
\]

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

\[
\text{bigor } \langle v,x,y \rangle \text{ in } [A,B,C],[1..9],[3..4] \text{ when } v \neq A \text{ and } x \neq y: \\
v(x)
\text{end}
\]

**Special cases for quantifier elimination**  Here is the list of “limit” cases where \texttt{bigand} and \texttt{bigor} will produce special results:

- In \texttt{bigand}, if a set is empty then \texttt{Top} is produced
- In \texttt{bigand}, if the \texttt{when} condition is always false then \texttt{Top} is produced
- In \texttt{bigor}, if a set is empty then \texttt{Bot} is produced
- In \texttt{bigor}, if the \texttt{when} condition is always false then \texttt{Bot} is produced

These behaviors come from the idea of quantification behind the \texttt{bigand} and \texttt{bigor} operators:

- Universal quantification  \( \forall x \in S, p(x) \) \texttt{bigand } \texttt{x in S: p(x) end}
- Existential quantification  \( \exists x \in S, p(x) \) \texttt{bigor } \texttt{x in S: p(x) end}
The following properties on quantifiers hold:

\[ \forall x \in \emptyset, p(x) \equiv \top \]
\[ \exists x \in \emptyset, p(x) \equiv \bot \]

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:

<table>
<thead>
<tr>
<th>Math notation</th>
<th>TouIST syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leq k ) ( x ) ( P )</td>
<td>atmost(( k ), ( P ))</td>
</tr>
<tr>
<td>( \geq k ) ( x ) ( P )</td>
<td>atleast(( k ), ( P ))</td>
</tr>
<tr>
<td>( &lt; &gt; k ) ( x ) ( P )</td>
<td>exact(( k ), ( P ))</td>
</tr>
</tbody>
</table>

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

- \( \leq k \) \( x \) \( P \) represents "at any time, at most \( k \) propositions \( x \in P \) must be true"
- \( \geq k \) \( x \) \( P \) represents "at any time, at least \( k \) propositions \( x \in P \) must be true"
- \( < > k \) \( x \) \( P \) 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, \( \text{exact}(5, p([1..20])) \) 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 \( \text{exact}(5, p([1..20])) \) is equivalent to

\( \text{atleast}(k, \{p | x \in P \}) \)

#### Example 1.

\( \text{exact}(2, \{a, b, c\}) \) is equivalent to

\[ (a \land b \land \neg c) \lor (a \land \neg b \land c) \lor (\neg a \land b \land c) \]

### 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.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( P )</th>
<th>Gives</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 0 )</td>
<td>( P = \emptyset )</td>
<td>Top</td>
</tr>
<tr>
<td>( k = 0 )</td>
<td>( P \neq \emptyset )</td>
<td>bigand ( $p ) in $P: not $p end</td>
</tr>
<tr>
<td>( k &gt; 0 )</td>
<td>(</td>
<td>P</td>
</tr>
<tr>
<td>( k = 1 )</td>
<td>any</td>
<td>Top</td>
</tr>
<tr>
<td>( k &gt; 0 )</td>
<td>( \emptyset )</td>
<td>Bot (subcase of next row)</td>
</tr>
<tr>
<td>( k &gt; 0 )</td>
<td>(</td>
<td>P</td>
</tr>
<tr>
<td>( k &gt; 0 )</td>
<td>(</td>
<td>P</td>
</tr>
<tr>
<td>( k = 0 )</td>
<td>( \emptyset )</td>
<td>Top (subcase of next row)</td>
</tr>
<tr>
<td>( k = 0 )</td>
<td>any</td>
<td>Top</td>
</tr>
</tbody>
</table>

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".

### 2.7.4. 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:
Chapter 2. Language reference

### 2.7.5. 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

### 2.7.6. 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):

\[
<\text{formula-qbf}> ::= 
\begin{align*}
& \text{
exists \ comma-list(<prop>|<var>)<for> \\
\text{forall} \ comma-list(<prop>|<var>)<for>
\end{align*}
\]

\[
<\text{for}> ::= \text{for} <\text{var}> \text{ in } <\text{set}>
\]

One quantifier can quantify over multiple propositions. For example:

\[
\forall e,d: (\exists a,b: a \Rightarrow b) \Rightarrow (e \text{ and } \forall c: e \Rightarrow c)
\]

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

**Free variables**

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

**Renaming during the prenex transformation**

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

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 an error.

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

\[
\exists x: x
\]

\[
x \Rightarrow y
\]
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

\((\text{exists } x: x) \land (x \Rightarrow y)\)

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

\begin{verbatim}
  exists x: x
  and (x => y)
\end{verbatim}

\textbf{Multiple exists or forall}

The keyword \texttt{for} used in \texttt{exists} or \texttt{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:

\begin{verbatim}
  exists a: exists b: exists c: exists d: phi ;; (1)
  exists $p$ for $p$ in [a,b,c,d]: phi ;; equivalent to (1)
\end{verbatim}

\textbf{2.7.7. Quoted formulas (formulas as expressions)}

When you want to use formulas in sets, you must use double quotes "..." to “protect” the formula from being evaluated as a normal expression (available with \texttt{TouIST >= 3.5.1}). For example:

\begin{verbatim}
  bigand $f$ in ["a or b", "c or d", "e"]: $f$
  end
\end{verbatim}

will give the formula

\[(a \lor b) \land (c \lor d) \land e\]

\textbf{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).

\textbf{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.

\begin{verbatim}
<touist-file> ::= <affect> <touist-file>  | <formula> <touist-file>  | EOF
<expr> ::= <int>|<float>|<prop>|<bool>|<set>
  | "="<formula-simple>"="  <- Touist >= 3.5.1
<var> ::= "$" TERM
  | "$" TERM "(" <comma-list(<expr>)> ")"
<prop> ::=  | <var>  | TERM
\end{verbatim}
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>
Chapter 2. Language reference

\[
<\text{formula-smt}> ::= <\text{formula-smt}>|<\text{expr-smt}>
\]

\[
<\text{expr-smt}> ::= "Top"|"Bot"|<\text{prop}|<\text{var}|<\text{int}|<\text{float}|
\]

\[
<\text{order}>(<\text{expr-smt}>)|<\text{num-operations-standard}>(<\text{expr-smt}>)|<\text{equality}>(<\text{expr-smt}>)|<\text{in-parenthesis}>(<\text{expr-smt}>)
\]

\[
<\text{formula-qbf}> ::= "Top"|"Bot"|<\text{prop}|<\text{var}|<\text{formula-qbf}>|"exists"<\text{comma-list}>","|"forall"<\text{comma-list}>","[for]:="<\text{formula-qbf}>
\]

\[
<\text{for}> ::= "for"<\text{var}|"in"<\text{set}>
\]

\[
<\text{comma-list}> ::= <\text{comma-list}>,<|<\text{prop}|<\text{var}>
\]
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

```bash
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).

![Diagram of touist and touist.jar](image)

3.2. Usage

Any touist command is of the form:

```bash
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).

The man page of touist is available through `man touist` or `touist --help`. It contains almost everything you need to know its features and arguments.
3.2.1. Exit status

<table>
<thead>
<tr>
<th>Code</th>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
<td>with <code>--solve</code> or <code>--solver</code>, means SAT</td>
</tr>
<tr>
<td>8</td>
<td>UNSAT</td>
<td>with <code>--solve</code> or <code>--solver</code>, means UNSAT</td>
</tr>
<tr>
<td>50</td>
<td>TRANSL_ERROR</td>
<td>any error of syntax, type or during the translation</td>
</tr>
<tr>
<td>100</td>
<td>SOLVER_ERROR</td>
<td>any solver error (memory overflow, wrong something...)</td>
</tr>
<tr>
<td>124</td>
<td>CLI_ERROR</td>
<td>something wrong with the command-line arguments</td>
</tr>
<tr>
<td>125</td>
<td>BUG</td>
<td>the solver did not find any model</td>
</tr>
<tr>
<td>9</td>
<td>SOLVER_UNKNOWN</td>
<td>Note that the UNSAT is not really an error, but we choose to return a non-zero exit status anyway.</td>
</tr>
</tbody>
</table>

3.2.2. Usage for propositional logic (SAT mode)

The language accepted for propositional logic is described in 2.7.4. This mode is enabled by default, but can be optionally expressed using the `--sat` flag.

Mapping table for DIMACS output

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 before the prelude line (i.e., `p cnf x y`; comments must be before the prelude line in the DIMACS specification). For example, if you run:

```bash
echo 'rain => wet_road rain not wet_road' | touist -
```

you will get the output:

```
c wet_road 1 <- mapping between DIMACS integers and propositions
c rain 2
p cnf 2 3 <- prelude of the DIMACS file
1 -2 0
2 0
-1 0
```

You can redirect this mapping table using the `--table <filename>` flag.

SAT solver

By default, `touist` embeds Minisat (see [6]), 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:

```bash
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:

```bash
echo a and b | touist - --solve | grep "1"
```

**--solve --interactive** allows the user to display the models one after the other. Press enter or any other key to continue and `q` or `n` to stop.
--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:

```bash
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 translate the given TouIST code to \LaTeX. Two options are available:

- with --latex (synonym of --latex=mathjax, 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=document, 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=AST outputs the AST or translation steps; AST can be - form shows the formula generated by the given TouIST file. This is useful for debugging and testing that the constructs bigand, bigor, exact... are correctly evaluated. - cnf shows the AST after the CNF transformation (warning: huge). - duringcnf shows the recursive translation steps leading to the CNF AST. - prenex and duringprenex are similar to the two previous ones but for prenex transformation (only with --qbf).

--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 \text{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.

--verbose[=N] or -v[N] (N defaults to 1) prints more information on timing and errors. With no N given (i.e., N=1):

- time spent on translation and solving are displayed (see table 12).
- on syntax error, print more AST context ((loc) for example).
- when an exception is raised, tries to show the stack trace.
- on syntax error, print 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.

With N \geq 2, --solver=CMD displays the stdin, stdout and stderr of CMD.

Note. Availability of timings using -v:

<table>
<thead>
<tr>
<th>---sat</th>
<th>---smt</th>
<th>---qbf</th>
</tr>
</thead>
<tbody>
<tr>
<td>translate</td>
<td>✓</td>
<td>(instantaneous)</td>
</tr>
<tr>
<td>--solve</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>--solver=</td>
<td>✓</td>
<td>(cmd not available)</td>
</tr>
</tbody>
</table>

3.2.4. Usage for Satisfiability Modulo Theory (SMT mode)

The language accepted by the SMT mode is described in 2.7.5.

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:

<table>
<thead>
<tr>
<th>LOGIC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_IDL</td>
<td>Integer Difference Logic</td>
</tr>
<tr>
<td>QF_RDL</td>
<td>Real Difference Logic</td>
</tr>
<tr>
<td>QF_LIA</td>
<td>Linear Integer Arithmetic</td>
</tr>
<tr>
<td>QF_LRA</td>
<td>Linear Real Arithmetic</td>
</tr>
</tbody>
</table>

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:

```bash
opam install qbf
```

Then, we can solve a small example:

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

which will give you a partial model:

```bash
? 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.

3.3. Using external solvers from _touist (--)solver"

Most SAT and QBF solvers accept the standardized DIMACS (resp. QDIMACS) as input language. You can give the DIMACS output of _touist_ directly to the solver and use the mapping table (see 3.2.2.1). But you can use _TouIST_ to do both the call to the solver as well as the translation of the resulting DIMACS model back to propositions names, using the argument --solver:

```bash
touist [--sat|--qbf] --solver="<cmd-and-arguments>" [--verbose]
```

For debugging purposes, you can add --verbose to see the stdin/stdout/stderr. The exit code of _touist_ is the same as with --solve.

The external solvers must use the following Minisat + (Q)DIMACS conventions: - should accept DIMACS or QDIMACS on standard input; - should print a model (or a partial model) in DIMACS on standard output; the model can span on multiple lines, each line begins with _v_, _V_ or nothing (for Minisat compatibility), and each line is optionally ended with 0.

```bash
v -1 2 -3 4 0
v 5 -6 0
```

• should return 10 (as error code) if problem is SAT, 20 if UNSAT.

Tested SAT solvers (_brew_ is available on _linux_ and _mac_):

- **minisat**

  ```bash
  brew install minisat
touist test/sat/sudoku.touist --solver="minisat /dev/stdin /dev/stdout"
  ```

- **picosat** (2015, version 965, SAT Race’15)

  ```bash
  brew install touist/touist/picosat
touist test/sat/sudoku.touist --solver="picosat --partial"
  ```

- **glucose** (2016, version 4.1, syrup is the parallel version)
brew install touist/touist/glucose
touist test/sat/sudoku.touist --solver="glucose -model"
touist test/sat/sudoku.touist --solver="glucose-syrup -model"

Tested QBF solvers:

- **caqe** (2017-07-19, CAQE qbfeval 2017, binary release without certification). Download the version *CAQE qbfeval 2017 (2017-07-19) binary release without certification* which will give you **caqe-mac**:
  touist test/qbf/allumettes2.touist --qbf --solver="/caqe-mac --partial-assignments"

  They also have a Homebrew tap repository but this version does not contain the needed **partial-assignments**.

- **qute** (2017-02-27, fork maelvalais/qute, minisat-based)
  brew install touist/touist/qute
touist test/qbf/allumettes2.touist --qbf --solver="qute --partial-certificate"

- **depqbf** (2017-08-02, DepQBF 6.03, Minisat-based QCDCL)
  
  brew install depqbf
touist test/qbf/allumettes2.touist --qbf --solver="depqbf --qdo --no-dynamic-nenofex"

- **quantor** (2014-10-26, Quantor 3.2). It is not necessary to use this solver externally as it is included with touist (see 3.2.5.1).
  
  brew install touist/touist/quantor
touist test/qbf/allumettes2.touist --qbf --solver="quantor"

- **rareqs** (2012, v1.1, CEGAR)
  
  brew install touist/touist/rareqs
touist test/qbf/allumettes2.touist --qbf --solver="rareqs"
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


