ToQ Overview

ToQ is a standalone or program-callable translator. It accepts a C-like syntax which may control a series of one or more assert statements. Enter "toq -a" for an overview of the syntax and semantics. The asserts act on logical entities (booleans) which define an optimization problem. The collection of run-time active asserts are transformed internally and presented to the D-Wave system, which finds the optimal solution to the input problem, and the results (values of the booleans) are passed back to the caller. There are about 300 help topics, which are reachable by entering "toq -H topic" or enter "toq -H help" for a full list of the help topics. Comments start with the "#" character and run to the end of the line.

ToQ accepts five types of statements:

- Directives - may appear anywhere in the program text.
  Use the command "listdirs:" to get the list of directives.
  Alternately, enter "toq -d" for a list of the directives, or
  "toq -D" for a list of the directives with definitions

- Declarations - variables must be declared before use, and
  all declarations must precede the first executable statement.
  Variable types are
  ```
  int:   32-bit integer values
  real:  64-bit floating point values
  intp:  32-bit integer values (post processing only)
  realp: 64-bit floating point values (post processing only)
  bool:  boolean entities (0,1) for use in assert statements
  mbool: multi-bit logical entities for use in assert
                statements
  bmask: multi-bit logical entities for use in assert
                statements
         All bool: names start with "@"
  ```

- Asserts - Executable statements that act on bool: variables
  They are interpreted at run-time, and the full sets of results
  are passed to the D-Wave co-processor for resolution. For
  more information about asserts, enter "toq -H assert:"
  Here are some samples
  ```
  bool:  @b1, @b2, @b3, @b4
  assert: And( @b1, Or(@b2,@b3) )
  assert: OneOf( @b2,@b4 )
  ```

- Control-flow - Executable statements which control the flow
  of execution. Here is the valid set
  ```
  if:    elseif:   else:   endif:
  ```

o Executables - simple assignment and control-flow statements which are interpreted and determine which assert statements are active. Executable statements may employ any of the 100+ functions and 60+ constants. Use "listfuncs:" and "listconsts:" to get lists of the functions and constants. Alternately, enter

"toq -f" for a list of the functions
"toq -F" for a list of the functions with definitions
"toq -c" for a list of the constants
"toq -C" for a list of the constants with definitions

Here are some samples

```
bool:  @b1, @b2, @b3, @b4, @b5
real:  press, temp, dilbert, omega
#~~~
omega = 34.5*PI
press = 744.3
temp = 199.3*Sqrt(press-150)
assert: Xor( @b2,@b5 )
assert: NOrMore( 1,@b1,@b3,@b5 )
if: (temp*Sin(omega*D2RADIANS) > 511)   # Very hot
        assert: And( @b5, Or(@b2,@b3) )
    else:   # Temp in normal range
        assert: TwoOf( @b1,@b3,@b5 )
        assert: And( @b2, Or(@b1,@b3) )
    endif:
assert: Xor( @b1,@b2 )
end:
```

Incidently, this code snippet would generate two warnings: ("@b4" and "dilbert" are never used). If "press" was less than 150, a run-time error would have been captured, and the execution would be terminated with a message back to the caller

ToQ syntax/semantics

o Input is case-insensitive (although input case is retained for output presentation).

o Blanks and tabs are ignored.

o Statements are terminated by the end of line.

o Variable names are 1-12 characters from (a-z, A-Z, 0-9), and the first character must be alphabetic. The only exception is that the first character of boolean and other constraint variables is "@" which is followed by 1-11 characters from (a-z, A-Z, 0-9).

o Variables may be the arguments to functions. Use "listops:" for a list of all the functions and their descriptions.

o Use "listconsts:" for a list of all the constants and their values and descriptions.

o All entities share a single name-space, so a variable may not be named "sin" or "pi."

o All declaration statements must appear before the first executable statement. (Asserts are executable.)

o Bool variables (which can take on only the values (0,1)), are not available for programmer use. They are used only in assert statements and their values are set by the run-time system such that all combinations of all the booleans are exercised. A ToQ program searches for one or more sets of the boolean values which make all the asserts TRUE.
<> Each ToQ program must contain 2 or more assert statements.
<> Each assert statement must contain 2 or more boolean variables.
<> Each boolean must appear in 2 or more "contributing" asserts. A contributing assert is one that can be evaluated as both TRUE and FALSE for different sets of boolean values. For example, given bools (@a,@b,@c), "assert: @a+@b+@c <= 2" may be either TRUE or FALSE, but "assert: @a+@b > 3" is always FALSE (this would be flagged as an input error).

o Bool variables (0,1) can be only used in assert statments. But note that they can utilized both arithmetically via {+,-,*,...} and logically via {&,|,>,<,...}. Groups of boolean can also be utilized - e.g.,
assert: (((@a + @b + @c) > 1) || (@x & @y))
See below (or enter "toq -H operator") to see a list of all the operators and the precedence rules.

o Assert statements can act only on bool: variables. When you use constraint variables in an assert statement, ToQ silently converts them to an expression built from hidden boolean variables. Use "listboolops" for a list of all the functions which accept bool: variables as arguments.

o The last statement should be "end:" - all statements after that will be ignored.

o When an error is detected during compile-time (e.g., mismatched parentheses), a message is generated and the error-scan continues, but no execution will occur. When an error is detected during run-time, a message is generated, and the

error scan may continue, but no further execution will occur. All error conditions are passed back to the caller.

- Enter "limits:" for a listing of the current ToQ limits (e.g., Max number of lines, max number of assert: statements, etc.)

- Enter "help: token" for a description of keyword "token," where token is a directive name, function name, constant name, or a keyword. "token" must be at least three characters.

- Enter "print: var1 [,...]" for a list of the current value of the listed variable name(s).

- Enter "printvars:" for a list of the current values of all the variables.

- A comment begins with the '#' character, which may appear anywhere on a line. The rest of the line is ignored (and thus the comment ends at the end of the line).

- All non-executable statements (except comments) must contain exactly one ":" character. Note that the assert statement and control-flow statements also contain exactly one ":" character

- All executable statements (except asserts and control-flow) must have exactly one replacement operator.

- The replacement operators are

```
=       +=      -=      *=      /=
&=      |=      >>=     <<=
```

  Note that the last four of these operators work only on integers.

- Executable statements, assert statements, and if statements may use comparison and logical operators.

- The comparison and logical operators are

```
==      !=      >       >=      ->
&&      ||      <       <=      <->
```

==	Equal	&&	And
!=	Not equal	||	Or
>	Greater than	<	Less than
->	Implies	<->	If-and-only-if
>=	Greater than or equal	<=	Less than or equal

  Note that "&&" and "||" work only on integers.

o The other available operators are

```
                +       -       *       /       ^       **
                &       |       !       ~       %       ^^
```

```
                   +  Add               &  Bitwise AND
                   -  Subtract          |  Bitwise OR
                   *  Multiply          !  Unary NOT
                   /  Divide            ~  Bitwise NOT
                   ^  Raise to power    %  Remainder
                   ** Raise to power    ^^ Bitwise XOR
```

o The operator precedence rules are:

Operator	Precedence	Operator	Precedence
(	1	>	8
)	1	<	8
Verb	2	>=	8
+ (unary)	3	<=	8
- (unary)	3	->	8
! (unyNot)	3	<-> (iff)	8
** (exp)	4	&	9
^ (exp)	4	|	9
*	5	^^ (xor)	9
/	5	&&	10
+	6	||	10
-	6	!=	11
<<	7	==	11
>>	7	+= -= *= /=	12
Operators of equal precedence resolve left to right		&= ^= |= %=	12
		>>= <<= =	12
		,	13

ToQ manages three levels of error conditions:

Warnings - messages are issued and execution continues

Syntax/Semantic Errors - messages are issued, and an error scan of the input continues, but execution is terminated

Run-time Errors - a message is issued, and an error-scan of the input may continue, but execution is terminated. A dump of the current values of all variables at the point of failure is generated.

It is important to note that when a (non-Warning) error condition is encountered, actual execution is stopped, and thus further error-

checking may be compromised. When reviewing output, one should always address the first error first, since that error may be masking or inducing errors further down. You may use the "-1" option to force ToQ to stop at the first error.

In all cases, ToQ returns the status to the calling program. When no errors are detected, ToQ returns the optimal solution to the input problem.

Constants, Directives, Functions

ToQ supports a host of everyday and special purpose constants and functions, as well as a group of directives which control execution. Enter: "toq -H item" to read about any item.

65 Constants

CUBERT2	LOG10E	MASK14	MASK27	PLANCK
CUBERT3	MASK02	MASK15	MASK28	PLANCK2PI
D2RADIANS	MASK03	MASK16	MASK29	RADIANS2D
ELECTRIC	MASK04	MASK17	MASK30	SQRT10
ENAT	MASK05	MASK18	MASK31	SQRT2
FOURTHPI	MASK06	MASK19	MASK32	SQRT3
HALFPI	MASK07	MASK20	MASK3201	SQRT5
JOSEPHSON	MASK08	MASK21	MASK3210	SQRT6
LN10	MASK09	MASK22	NUCLMAGN	SQRT7
LN2	MASK10	MASK23	PHI	SQRT8
LN3	MASK11	MASK24	PHIINV	SQRTE
LOG102	MASK12	MASK25	PI	SQRTPI
LOG103	MASK13	MASK26	PI43	THIRDPI

52 ToQ Directives

assert1:	listboolfuncs:	post:
assert2:	listbools:	postio:
assert:	listcode:	postm:
assertp:	listconstdefs:	print:
cladesoff:	listconsts:	printf:
conffile:	listdirdefs:	printfn:
dbprintb:	listdirs:	printsummary:
dbprintf:	listfuncdefs:	printvars:
dbquit:	listfuncs:	prototype:
end:	listgen:	satisfiability:
epilogue:	listops:	syntax:

```
errormgmt:    listprotos:      timing:
flipflop:     listsatdefs:     varfile:
force:        listsats:        viewvectors:
help:         maximize:        warningsoff:
initfile:     minimize:        whereami:
limits:       overview:        xrefoff:
listall:
```

113 Functions

```
Abs(x)                  IsInt(x)                Permutations(m,n)
Acos(x)                 LCM(m1,m2,...)          PopCount(n)
Acosh(x)                Ln(x)                   Prime(n)
All(b1,b2,...)          Ln1P(x)                 Rand(a,b)
Amalgam(b1,b2,...)      Log(x)                  RandSeed(n)
And(b1,b2)              Log2(x)                 Round(x)
Antilog(x)              Lucas(n)                Same(m1,m2,...)
Any(b1,b2,...)          MaskN(N)                Sat(b1,...)
Asin(x)                 Max(x1,x2,...)          Sat12(b1,b2)
Asinh(x)                MaxAbs(x1,x2,...)       Sat13(b1,b2,b3)
Atan(x)                 Min(x1,x2,...)          Sat14(b1,b2,b3,b4)
Atan2(x,y)              MinAbs(x1,x2,...)       Sat15(b1,b2,b3,b4,b5)
Atanh(x)                Mod(j,k)                Sat2(b1,b2)
Bitcount(n)             NAE3Sat(b1,b2,b3)       Sat23(b1,b2,b3)
Bitmask(b1,m,n1,n2,...) NAE4Sat(b1,b2,b3,b4)    Sat24(b1,b2,b3,b4)
Ceiling(x)              NAE5Sat(b1,b2,b3,b4,b5) Sat25(b1,b2,b3,b4,b5)
Combinations(m,n)       NAESat(b1,b2,...)       Sat3(b1,b2,b3)
Cos(r)                  NAESat3(b1,b2,b3)       Sat34(b1,b2,b3,b4)
Cosh(x)                 NAESat4(b1,b2,b3,b4)    Sat35(b1,b2,b3,b4,b5)
Cubert(x)               NAESat5(b1,b2,b3,b4,b5) Sat4(b1,b2,b3,b4)
Diff(m1,m2,...)         NAESatN(N,b1,b2,...)    Sat45(b1,b2,b3,b4,b5)
Even(n)                 Nand(b1,b2)             Sat5(b1,b2,b3,b4,b5)
Exp(x)                  NBitHit(b1,m,n1,n2,...) SatN(N,b1,b2,...)
Expm1(x)                NDiff(x1,x2,...)        Sin(r)
Factorial(n)            NearInt(x,eps)          Sinh(x)
Fibonacci(n)            NOf(N,b1,b2,...)        Sqrt(x)
Floor(x)                None(b1,b2,...)         SqrtAbs(x)
FormalMod(a,b)          Nor(b1,b2)              Sum(n)
GCD(m1,m2,...)          NOrLess(N,b1,b2,...)    SumDelta(a,d,n)
GSeries(m1,m2,n)        NOrMore(N,b1,b2,...)    Tan(r)
IFF(b1,b2)              Not(b1)                 Tanh(r)
Implies(b1,b2)          NXor(b1,b2)             Trunc(x)
InRangeEq(x,min,max)    Odd(n)                  TwoOf(b1,b2,...)
InRangeNEq(x,min,max)   OneOf(b1,b2,...)        XNor(b1,b2)
InSet(val,x1,x2,...)    Or(b1,b2)               Xor(b1,b2)
Inv(x)                  OutRangeEq(x,min,max)   Xor2Sat(b1,b2)
InvMod(j,k)             OutRangeNEq(x,min,max)  Xor3Sat(b1,b2,b3)
IRand(j,k)              OutSet(val,x1,x2,...)
```

ToQ commandline

ToQ - a simple program to (conditionally) build QUBOs and
execute them on a D-Wave Quantum Computing System.

ToQ implements a simple environment for assessing and evaluating assertions about the permissible values for a set of binary decision variables. The input "program" may be from stdin or a file, and the output goes to stdout or a file. Configuration information is available from your environment (see the qOp's dw command) or from a file. Enter:

```
toq  -H environment   -or-
toq  -H files
```

for details. Users may provide a file with variable/value pairs for use at startup and/or a file with configuration information. Both files use the -z option (see -Z for file formats).

Typical usages:

```
1) toq  -i xyz.toq  # PreQuantum Analysis only
2) toq  -i xyz.toq  -r  -z ConfFile  -o OutFile
3) toq  -i xyz.toq  -r  -z ConfFile  -z VarFile
4) toq  -i xyz.toq  -q  -T
5) toq  -i xyz.cnf  -r  -z ConfFile
6) toq  -i xyz.qubo
7) toq  -i xyz.qbout
```

- Case 1 runs only the FrontEnd (parsing, error detection, assert report, Xref map, clade analysis, ...), and does not solve the problem. Output is written to stdout.
- Case 2 runs the full program, uses ConfFile for configuration data, and writes output to OutFile
- Case 3 runs the full program, and reads incoming variable/value pairs from VarFile
- Case 4 runs the full program, uses the Qbsolv (-q) program, and reports timings (-T) for each step
- Case 5 (with a .cnf file) is a special case with xyz.cnf containing a standard DIMACS-format conforming satisfiability problem (enter "toq -H cnf" for the format of a .cnf file).
- Case 6 accepts a .qubo file and works on a general unconstrained binary optimization file (enter "toq -H .qubo" for the format of a .qubo file). The default intermediate output file for this case is xyz.qbout, and ToQ output is presented on stdout.
- Case 7 accepts xyz.qbout as the output file from Case 6 and presents the results (produces the same output as Case 6).

```
The vast majority of ToQ calls take this form:
            toq  -r  -i xyz.toq
There are a large variety of special options to meet various needs
(as the results below show), but beginners and indeed most users
rarely need to use them.

Usage: toq  [-abBcCdDeEfFg] [-G fmt] [-h] [-H topic] [-i inFile]
             [-o outFile] [-O opt(s)] [-Lpq] [-Q opt] [-rRsS]
             [-t satType] [-TuvxXZ] [-z varFile|configFile] [-Z]

    -a  Syntax/semantics overview (no execution)

    -b  List the bool- and constraint-eligible functions
        (no execution)

    -B  List the bool- and constraint-eligible functions
        w/ descriptions  (no execution)

    -c  List the constants (no execution)

    -C  List the constants w/ values, descrs (no execution)

    -d  List the directives (no execution)

    -D  List the directives w/ descrs (no execution)

    -e  List the error handling attributes (no execution)

    -E  Do not show constraint variable declaration error hints

    -f  List all the functions (no execution)

    -F  List all the functions w/ descriptions (no execution)

    -g  List the generated boolVariables and assertStmts

    -G  Group result display format for .qubo input files.
        Followed by a string (e.g., toq -i zz.qubo -G 3,ABCDEFGH
        -or-  toq -i ww.qubo -G 4).  Defines various output
        display formats (enter "toq -QG" for details and examples)

    -h  Help - print this message (no execution)

    -H  Help topic - print help message about "topic" (no exec)

    -i  The input file name (default is stdin)
            Typical case:  toq  -i xyz.toq
            Typical case:  toq  -i pqr
            Special Sat(DIMACS) case:  toq  -i abc.cnf
```

```
        Special Qubo case:  toq  -i def.qubo

-L  List the input program (after scan)

-n  Number of results requested (overrides env, startup files)
        Default: 500, Range: (1-2,000) is silently enforced

-o  The output file name (default is stdout)

-O  Options(s)Print:  Show detailed help for one or more
    options.  E.g., "toq -Ox" describes the -x option
    (no execution)

-p  Print all variable values at completion

-q  Use the direct qubo method to solve problem (via qbsolv).
    For input = xyz.toq, 2 files will be generated:
        xyzTOQ.qubo and xyzTOQ.qbout

-Q  Special options (defined by next argument)
        -Q afile - "file" is Quantum Apprentice file, nolist
        -Q Afile - "file" is Quantum Apprentice file, list
        -Q c - Collect all (cnf) single var msgs into
               1 warning
               (Default behavior is an error msg)
        -Q C - Disable all (cnf) single var msgs
               (This option is not recommended)
        -Q D - Debug mode
        -Q F - Disable Clade (assert/family) Report
               (This option is not recommended)
        -Q G - Show details/examples for -G option (no exec)
        -Q M - Disable mult bool/assert msg (not recommended)
        -Q P - Enable Partitioning
        -Q Q - Use the Quantum Simulator Solver
        -Q R - Regression testing mode
        -Q Se - Synopsis - Embedded ToQ (no execution)
        -Q Sm - Synopsis - Programming Model (no execution)
        -Q Sp - Synopsis - Partitioning (no execution)
        -Q Sq - Synopsis - Full QUBO processing
                (no execution)
        -Q V - View generated vectors
        -Q W - Disable Warning messages (not recommended)
        -Q Y - Disable FYI messages

-r  Run the BackEnd and display D-Wave stats

-R  Run the BackEnd

-s  List the Satisfiability functions (no execution)
```

```
      -S  List the Satisfiability functions w/ descriptions
          (no execution)

      -t  Type of satisfiability clauses (for "cnf" only,
          (not "cnf+")).  Valid strings for "satType" are in
          { 1in3, 2in3, 2in4, 2in5, 3in4, 3in5, 4in5, nae, nae3,
            nae4, nae5, naen, sat2, sat3, sat4, sat5, xor2, xor3 }
             Simple example:  toq  -i xyz.cnf  -t nae3

      -T  Show detailed timing data

      -u  User manual (list options combined - no execution)

      -v  Print Version information - no execution)

      -x  Disable the ConstraintVar/AssertId Xref

      -X  Suppress all output but errors, warnings, Summary
          (Not for new (programs, programmers))

      -z  AuxFileName - read/process this file.  The auxiliary
          files are VariablesIn and Configuration (see -Z)

      -Z  Format (layout) of the VariablesIn, Configuration,
          and .cnf files (no execution)

      -1  Stop at 1st error detected (default: OFF)

-a  Syntax/semantics overview'    -o  Output file name
-b  List bool/constraint funcs'   -O  Option(s)Print'
-B  List bool/constr funcs,descr' -p  Print var values at end
-c  List constants'               -q  Use Qbsolv to solve problem
-C  List constants w/ descr'      -Q  Special options
-d  List directives'              -r  Run the BackEnd, show stats
-D  List directives w/ descr'     -R  Run the BackEnd
-e  List error handling attrs'    -s  List Sat functions'
-E  Don't show constr hints       -S  List Sat funcs w/ descr'
-f  List functions'               -t  Type Sat clauses'
-F  List functions w/ descr'      -T  Show detailed timing
-g  List gen'd bools/asserts      -u  List user manual'
-G  Group result display format'  -v  Version info'
-h  Help (show cmdline options)   -x  Disable Xref
-H  Help topic (help: "topic")'   -X  Suppress non-error,-warnings
-i  Input file name               -z  AuxFileNm (vals,config)
-L  List input program            -Z  Format (layout) of files'
-n  Number of results             -1  Stop at 1st error
-o  Output file name
                                                ' no execution
                                ~~~\\~~~
```