MBT

TLA+ Cheatsheet

TLA+ Quick Start - contains enough to model almost anything

```
(* Comments *)

(* This is
   multiline comment *)
\* This is single line comment

(* Module structure *)

---- MODULE <module> ----   \* Starts TLA+ module (should be in file <module>.tla)
====                        \* Ends TLA+ module (everything after that is ignored)

EXTENDS <module>            \* EXTEND (import) another TLA+ module
VARIABLES x, y, ...         \* declares variables x, y, ...
CONSTANTS x, y, ...         \* declares constants x, y, ... (should be defined in configuration

Name == e                   \* defines operator Name without parameters, and with expression e
Name(x, y, ...) == e        \* defines operator Name with parameters x, y, ..., and body e (may

(* Boolean logic *)

BOOLEAN                     \* the set of all booleans (same as {TRUE, FALSE})
TRUE                        \* Boolean true
FALSE                       \* Boolean false
~x                          \* not x; negation
x /\ y                      \* x and y; conjunction (can be also put at line start, in multi-li
x \/ y                      \* x or y; disjunction (can be also put at line start, in multi-lin
x = y                       \* x equals y
x /= y                      \* x not equals y
x => y                      \* implication: y is true whenever x is true
x <=> y                     \* equivalence: x is true if and only if y is true

(* Integers *)              \* EXTENDS Integers (should extend standard module Integers)

Int                         \* the set of all integers (an infinite set)
1, -2, 1234567890           \* integer literals; integers are unbounded
a..b                        \* integer range: all integers between a and b inclusive
x + y, x - y, x * y         \* integer addition, subtraction, multiplication
x < y, x <= y               \* less than, less than or equal
x > y, x >= y               \* greater than, greater than or equal

(* Strings *)

STRING                      \* the set of all finite strings (an infinite set)
"", "a", "hello, world"     \* string literals (can be compared for equality; otherwise uninter

(* Finite sets *)           \* EXTENDS FiniteSets (should extend standard module FiniteSets)

{a, b, c}                   \* set constructor: the set containing a, b, c
```

```
Cardinality(S)              \* number of elements in set S
x \in S                     \* x belongs to set S
x \notin S                  \* x does not belong to set S
S \subseteq T               \* is set S a subset of set T? true of all elements of S belong to
S \union T                  \* union of sets S and T: all x belonging to S or T
S \intersect T              \* intersection of sets S and T: all x belonging to S and T
S \ T                       \* set difference, S less T: all x belonging to S but not T
{x \in S: P(x)}             \* set filter: selects all elements x in S such that P(x) is true
{e: x \in S}                \* set map: maps all elements x in set S to expression e (which may

(* Functions *)

[x \in S |-> e]             \* function constructor: maps all keys x from set S to expression e
f[x]                        \* function application: the value of function f at key x
DOMAIN f                    \* function domain: the set of keys of function f
[f EXCEPT ![x] = e]         \* function f with key x remapped to expression e (may reference @,
[f EXCEPT ![x] = e1,        \* function f with multiple keys remapped:
          ![y] = e2, ...]   \*   x to e1 (@ in e1 will be equal to f[x]), y to e2 (@ in e2 will
[S -> T]                    \* function set constructor: set of all functions with keys from S

(* Records *)

[x |-> e1, y |-> e2, ...]   \* record constructor: a record which field x equals to e1, field y
r.x                         \* record field access: the value of field x of record r
[r EXCEPT !.x = e]          \* record r with field x remapped to expression e (may reference @,
[r EXCEPT !.x = e1,         \* record r with multiple fields remapped:
          !.y = e2, ...]    \*   x to e1 (@ in e1 is equal to r.x), y to e2 (@ in e2 is equal t
[x: S, y: T, ...]           \* record set constructor: set of all records with field x from S,

(* Sequences *)             \* EXTENDS Sequences (should extend standard module Sequences)

<<a, b, c>>                 \* sequence constructor: a sequence containing elements a, b, c
s[i]                        \* the ith element of the sequence s (1-indexed!)
s \o t                      \* the sequences s and t concatenated
Len(s)                      \* the length of sequence s
Append(s, x)                \* the sequence s with x added to the end
Head(s)                     \* the first element of sequence s

(* Tuples *)

<<a, b, c>>                 \* tuple constructor: a tuple of a,b,c (yes! the <<>> constructor i
                            \* - sequence elements should be same type; tuple elements may have
t[i]                        \* the ith element of the tuple t (1-indexed!)
S \X T                      \* Cartesian product: set of all tuples <<x, y>>, where x is from S

(* Quantifiers *)

\A x \in S: e               \* for all elements x in set S it holds that expression e is true
\E x \in S: e               \* there exists an element x in set S such that expression e is tru

(* State changes *)

x', y'                      \* a primed variable (suffixed with ') denotes variable value in th
UNCHANGED <<x,y>>           \* variables x, y are unchanged in the next state (same as x'=x /\

(* Control structures *)

LET x == e1 IN e2           \* introduces a local definition: every occurrence of x in e2 is re
IF P THEN e1 ELSE e2        \* if P is true, then e1 should be true; otherwise e2 should be tru
```

Apalache

```
# Running apalache

# A handy alias for calling Apalache
alias apalache="java -jar apalache-pkg-0.17.5-full.jar --nworkers=8"

# Typecheck
apalache typecheck <.tla file>

# Model check assuming a .cfg file with the same name as the .tla file is present
apalache check <.tla file>

# Model check assuming with a specific .cfg file
apalache check --config=<.cfg file> <.tla file>

# Model check an invariant Foo
apalache check --inv=Foo <.tla file>

# Generate multiple (up to n) traces for invariant Foo
apalache check --view=<View Operator Name> --max-error=n --inv=Foo <.tla file>
```

```
(* Writing models with Apalache *)

EXTENDS Apalache       \* Import https://github.com/informalsystems/apalache/blob/unstable/src/

\* Makes Apalache understand that a function with keys in 1.maxSeqLen can be treated as a Seque
FunAsSeq(fn, maxSeqLen) == SubSeq(fn, 1, maxSeqLen)

(*
Equivalent to the pseudocode:

x = initialValue
for element in arbitrary_ordering(S):
   x = CombinerFun(x, element)
return x

*)
FoldSet(CombinerFun, initialValue, S) \* For a set S

(*
Equivalent to the pseudocode:

x = initialValue
for element in sequential_ordering(S):
   x = CombinerFun(x, element)
return x

*)
```

```
FoldSeq(CombinerFun, initialValue, S) \* For a sequence S
```

TLC

```
# Running TLC

# A handy alias providing the JVM with 12GB of RAM (adjust accordingly) and using multiple thr
alias tlc="java -XX:+UseParallelGC -Xmx12g -cp tla2tools.jar tlc2.TLC -workers auto"

# Model check with TLC
tlc -config <.config file> <.tla file>

# Run TLC in simulation mode
tlc -config <.config file> -simulate <.tla file>
```

F.A.Q.

1. Does whitespace matter in TLA+?

Yes whitespace matters when AND'ing (/\), OR'ing (\/)

```
Foo == /\ x
       /\ \/ y
          \/ z
```

means

```
Foo == x and ( y or z)
```

Indentation defines precedence and must be tab or space aligned. The parser should complain if it not aligned.

2. .... coming soon.