State Machine

A state machine is described using three things:

1. The **variables** to be used,
2. The **initial values** of those variables, and
3. The **next states** that can follow any given state.

Execution

An **execution** of a system is described as a sequence of discrete steps.

Step

A **step** is a change from one state to another.

Execution + step

Thus an **execution** is represented as a sequence of states, where a **step** is the change from one state to the next

State

A **state** is nothing more than an assignment of values to all of the variables we defined.

ex. If we have a system with only two variables, **x** and **y**, then **[x:4, y:7]** is one state of the system, and **[x:25, y:6]** is a different state of the system

Behaviour

A **behaviour** is a sequence of states. Thus an **execution** is represented as a **behaviour.**

Deadlock

Deadlock is when a system permanently stops but it was supposed to keep going (i.e., deadlock = bad).

Termination

Termination is when a system stops, when it was intended to stop.

Invariant

Something that must be true in **every** state of every valid behaviour of a system.

Primed variable

A variable of the form **x'**, where the apostrophe means "primed". This indicates the value of the variable in the **next state**.

`x' = x + 1`	means "the value of **x** in the next state is the value of **x** in the current state plus 1

Enabling condition

The parts of a formula containing no primed variables. These parts are used to indicate "what must be true" for this action to be enabled, i.e the specify the valid "current states" of a system

```
/\ x > 5
/\ y \in 3..10
/\ x' = 30
/\ y' = 40
```

The first two lines above are the enabling conditions for this formula. The formula as a whole states: "If in the current state, x > 5 and y is a valid in the set {3,4,5,6,7,8,9,10}, then in the next state x will be 30 and y will be 40

Action

Any formula with primed variables is an **action**. Thus an **action** is a formula describing a possible state pair.

$\wedge$ Logical "AND". Called a "conjunct". Written in ASCII as `/\`

Written in Python as **`and`**. Written in C as **`&&`**

$\vee$ Logical "OR". Called a "disjunct". Written in ASCII as `\/`

Written in Python as **`or`**. Written in C as `||`

$\{\}$ The "empty set"

{1, 2, "a", "bar", {"foo"}} A set containing five elements

{1,2} = {2,1} Order does not matter in sets, these two sets are equal

$\in$ The "in set" operator. Written in ASCII as **`\in`**

"a" $\in$ {"a", "b"} is TRUE

Python: `"a" in set(["a", "b"])`

"a" $\in$ {"b", "c"} is FALSE

Python: `"a" in set(["b", "c"])`

$\subseteq$ The "subset" operator. Written in ASCII as **`\subseteq`**

{"a"} $\subseteq$ {"a", "b", "c"} is TRUE

Python: `set(["a"]).issubset(set(["a","b","c"]))`

{"a", "b"} $\subseteq$ {"b", "c"} is FALSE

Python: `set(["a","b"]).issubset(set(["b","c"]))`

∪ The “union” operator, written as **\union** in ASCII. It creates a new set by “squishing together” elements of two sets.

{“a”, “b”} ∪ {} = {“a”, “b”}

{“a”, “b”} ∪ {“c”, “d”} = {“a”, “b”, “c”, “d”}

{“a”, “b”} ∪ {“b”, “c”} = {“a”, “b”, “c”}

Python: `set([“a”,”b”]).union(set([“b”, “c”]))`

∩ The “intersect” operator, written as `\intersect` in ASCII. It returns a new set containing the elements common to both sets.

{“a”, “b”} ∩ {} = {}

{“a”, “b”} ∩ {“c”, “d”} = {}

{“a”, “b”} ∩ {“b”, “c”} = {“b”}

Python:

`set([“a”, “b”]).intersection(set([“b”, “c”]))`

.. Roughly equivalent to Python’s **range()** function.

1..10 = {1,2,3,4,5,6,7,8,9,10}

In Python: `set(range(1,11))`

∃ This means “there exists”, written as **\E** in ASCII.

The usual form is $\exists\ x \in S : P(x)$

This means “there exists some x in the set S such that P(x) is TRUE”

The entire expression evaluates to TRUE or FALSE

$\exists\ x \in \{1,2,3,4\} : x > 3$ is TRUE

$\exists\ x \in \{1,2,3,4\} : x > 5$ is FALSE

$\exists\ x \in \{1,2,3,4,5\} : x > 5$ in Python:

```
def exists(S):
  for x in S:
    if x > 5:
      return True
  return False

exists(set([1,2,3,4,5]))  # False
```

$\forall$ Means “for all”, written as **\A** in ASCII.

The usual form is $\forall\, x \in S : P(x)$

This means “for all x in the set S, it is the case that $P(x)$ is TRUE”

The entire expression evaluates to TRUE or FALSE

$\forall\, x \in \{1,2,3,4\} : x > 3$ is FALSE

$\forall\, x \in \{1,2,3,4\} : x > 0$ is TRUE

$\forall\, x \in \{1,2,3,4,5\} : x > 5$ in Python is:

```
def for_all(S):
  for x in S:
    if not (x > 5):
      return False
  return True

for_all(set([1,2,3,4,5]))  # False

# Equivalent to this as well
all(map(lambda x: x > 5, [1,2,3,4,5]))
```

$=$ Means “equality”. It is NOT an assignment operator. It is a Boolean operator. It is the = you would have learned in grade 1 mathematics.

$\triangleq$ Means “defined to be”. It is written as **==** in ASCII.

{ x $\in$ S : P } is like a **`filter()`** function, creating a new set consisting of the elements of **S** that satisfy **P.**

{ x$\in${1,2,3,4,5} : x > 3 } = {4,5}

This is equivalent to the following Python expressions:

```
set([x for x in [1,2,3,4,5] if x > 3])

set(filter(lambda x: x > 3, [1,2,3,4,5]))
```

{ e : x $\in$ S } is like a **`map()`** function, applying expression **e** to every element of **S.**

{ x * 2 : x$\in${1,2,3,4,5}} = {2,4,6,8,10}

This is roughly equivalent to the following Python expressions:

```
set([x*2 for x in [1,2,3,4,5]])

set(map(lambda x: x*2, [1,2,3,4,5]))
```

Functions

A TLA+ function is a *true* mathematical function. The term “function” we use in most programming languages is completely wrong. Those things should generally be called “sub-routines”

A TLA+ function is a set of **key**:**value** pairs. Kind of like an array, kind of like a Python dict or a Ruby hash

All **keys** in a function must be of the same type. i.e. they all must be numbers, or they all must be strings, or they all must be sets.

The **values** can be of mixed types

The set of **keys** are called the **DOMAIN**; the set of **values** are the **RANGE** or **IMAGE**

[i ∈ S |-> e]

This creates a function whose domain is all the elements of **S** (i.e., the keys are the elements of **S**), and the corresponding values are created by evaluating the expression **e**

Read this as: “Take the set to the left of **|->** and use that as the keys. For each key, evaluate the expression to the right of **|->**, and use that as the value”

[i ∈ {2,3,4} |-> i*2] = (2 :> 4 @@ 3 :> 6 @@ 4 :> 8)

That syntax on the right side of the = is rarely used. But it’s equivalent to the following Python dictionary: {2:4, 3:6, 4:8}, where TLA uses **:>** as the separator between a key and a value, and **@@** in place of commas. (**@@** is really “concatenating” or “merging” two functions into one)

[i ∈ {2,3,4} |-> i*2] is equivalent to the following Python dictionary comprehension `{i: i*2 for i in [2,3,4]}`

The value for some key can be accessed using the same syntax as array/dictionary access in Python

```
f == [ x \in {1,2,3} |-> x * 5 ]

f[1] = 5
f[2] = 10
f[3] = 15
```

DOMAIN

The **DOMAIN** operator returns the domain of a function, as a set

DOMAIN [i ∈ {2, 4, 6} |-> i*2] = {2,4,6}

[S -> T]

Creates a set of functions, where **S** and **T** are also sets

NOTE: The arrow there -> is different than the |-> arrow we just saw.

```
[{2,3} -> {"a", “b”, “c”}] =  { (2 :> "a" @@ 3 :> "a"),
                                (2 :> "a" @@ 3 :> "b"),
                                (2 :> "a" @@ 3 :> "c"),
                                (2 :> "b" @@ 3 :> "a"),
                                (2 :> "b" @@ 3 :> "b"),
                                (2 :> "b" @@ 3 :> "c"),
                                (2 :> "c" @@ 3 :> "a"),
                                (2 :> "c" @@ 3 :> "b"),
                                (2 :> "c" @@ 3 :> "c") }
```

That output is **roughly** equivalent to the following Python set of dictionaries:

```
{ {2:”a”, 3:“a"},
  {2:”a", 3:“b"},
  {2:”a", 3:“c"},
  {2:”b", 3:“a”},
  {2:”b", 3:“b"},
  {2:”b", 3:“c"},
  {2:”c", 3:“a"},
  {2:”c", 3:“b"},
  {2:”c", 3:“c”} }
```

<<“a”, “b”>>

The **<<>>** operator is used for tuples

Tuples are 1-based, unlike most programming languages which are 0-based

Elements of tuples can be accessed via “normal” programming language index notation

<< “a”, “b”, 42 >>[2] = “b”

Tuples are just syntactic sugar for functions! The domain is the set of integers 1..N, and the range is the values inside the tuple

EXCEPT

The **EXCEPT** operator is used to create a new function from an existing function, with certain values replaced

```
f == [x \in {2,4,6} |-> x*4]

t == [f EXCEPT ![4] = 30]
```

This says “create a new function **t**, which is like **f**, except the value at key **4** has been replaced with **30**

A Python equivalent:

```
import copy
f = {2: 8, 4: 16, 6: 24}
t = copy.deepcopy(f)
t[4] = 30
```

Records

Records are another syntactic sugar on top of functions, for cases where you want all the keys to be strings

Their syntax is **`[ key1 |-> value1, key2 |-> value2]`**, where **key1**, **key2,** …, **keyN** are literals

```
r == [nodes |-> {1,2}, edges |-> {“a”, “b”}, cost |-> 5]
r.nodes = {1,2}
r.edges = {“a”, “b”}
r.cost = 5
r[“edges”] = {“a”, “b”}
```

Equivalent to the following Python dictionary

```
r = {“nodes”: set([1,2]), “edges”: set([“a”, “b”]), “cost”: 5}
```

[key1: S, key2: T]

This creates a set of records, where each record has the same keys, but differing values

```
[nodes: {1,2}, edges: {“a","b","c"}]    =   { [nodes |-> 1, edges |-> "a"],
                                               [nodes |-> 1, edges |-> "b"],
                                               [nodes |-> 1, edges |-> "c"],
                                               [nodes |-> 2, edges |-> "a"],
                                               [nodes |-> 2, edges |-> "b"],
                                               [nodes |-> 2, edges |-> "c"] }
```

This is equivalent to the following Python:

```
set_of_records = set()
for node in [1,2]:
    for edge in [“a”,”b”,”c”]:
        set_of_records.add({“nodes”: node, “edges”, edge})


{  {“nodes”: 1, “edges”: ”a”},
   {“nodes”: 1, ”edges”: “b"},
   {“nodes”: 1, ”edges”: “c"},
   {“nodes”: 2, ”edges”: “a"},
   {“nodes”: 2, ”edges”: “b"},
   {“nodes”: 2, ”edges”: “c"} }
```