

## Comments

```
// one line
/* multiple
   lines */
```

## Basic types

**bool** – Booleans  
**int** – signed big integers  
**str** – string literals  
**type Name = otherType**  
type alias, starts with  
upper-case

## Literals

**false true**

**123 123\_000 0x12abcd**

**"Quint"**: str, a string

**Int**: Set[int] – all integers

**Nat**: Set[int] – all non-  
negative integers

**Bool** = Set(false, true)

## Records

**{ name: str, age: int }**  
record type

**{ name: "TLA+", age: 33 }**  
new record of two fields

**R.name** the field value

**R.with("name", "Quint")**  
copy of R but with the field  
set to the new value

**fields(R): Set[str]**  
the set of field names

## Basic algebraic data types

under construction 

## Sets - core data structure!

**Set[T]** – type: set with  
elements of type T

**Set(1, 2, 3)** – new set,  
contains its arguments

**1.to(4)** – new set:  
Set(1, 2, 3, 4)

**1.in(S)** – true, if the  
argument is in S

**S.contains(1)** – the same

**S.subseteq(T)** – true, if  
all elements of S are in T

**S.union(T)** – new set:  
elements in S or in T

**S.intersect(T)** – new set:  
elements both in S and in T

**S.exclude(T)** – new set:  
elements in S but not in T

**S.map(x => 2 \* x)** – new  
set: elements of S are  
transformed by expression

**S.filter(x => x > 0)** –  
new set: leaves the elements  
of S that satisfy condition

**S.exists(x => x > 10)** –  
true, if some element of S  
satisfies condition

**S.forall(x => x <= 10)** –  
true, if all elements of S  
satisfy condition

**size(S)** – the number of elements in  
S, unless S is infinite (Int or Nat)

**isFinite(S)** – true, if S is finite

**Set(1, 2).powerset()** all subsets:  
Set(Set(), Set(1), Set(2), Set(1, 2))

**flatten(S)** – union of all sets in S

**chooseSome(S)** – an element of S via a fixed rule

**S.fold(i, (s, x) => s + x)** go over elements of  
S in some order, apply the expression, continue  
with the result; i is the initial value of s

## Maps - key/value bindings

**a -> b** – type: binds keys of  
type a to values of type b

**Map(1 -> 2, 3 -> 6)** – binds  
keys 1, 3 to values 2, 6

**S.mapBy(x => 2 \* x)** – binds  
keys in S to expressions

**M.keys()** – the set of keys

**M.get(key)** – get the value  
bound to key

**M.set(k, v)** – copy of M: but  
binds k to v, if k has a value

**M.put(key, v)** – copy of M:  
but (re-)binds k to v

**M.set(k, (old => old + 1))**  
as M.set(k, v) but v is  
computed via anonymous  
operator with *old* == M.get(k)

**S.setOfMaps(T)** – new set:  
contains **all maps** that bind  
elements of S to elements of T

**Set((1, 2), (3, 6)).setToMap()**  
new map: bind the first elements of  
tuples to the second elements

## Tuples

**(str, int, bool)**  
tuple type

**("Quint", 2023, true)**  
new tuple

**T.\_1 T.\_2 T.\_3**  
get tuple elements

**tuples(S1, S2, S3)**  
the set of all tuples with  
elements in S1, S2, S3

## Lists - use Set, if you can

**List[T]** – type: list with  
elements of type T

**[1, 2, 3]** – new list, contains  
its arguments in order

**List(1, 2, 3)** – the same

**range(start, end)** – new list  
[start, start + 1, ..., end - 1]

**length(L)** – the number of  
elements in the list L

**L[i]** – ith element,  
if 0 <= i < length(L)

**L.concat(K)** – new list:  
start with elements of L,  
continue with elements of K

**L.append(x)** – new list:  
just L.concat([x])

**L.replaceAt(i, x)** – L's copy  
but the ith element is set to x

**L.slice(s, e)** – new list:  
[L[s], ..., L[e - 1]]

**L.select(x > 5)** – new list:  
leaves the elements of L that  
satisfy condition

**L.foldl(i, (s, x) => x + s)**  
go over elements of L in order,  
apply expression, continue with  
the result; i is the initial  
value of s

**L.foldr(i, (x, s) => x + s)**  
as foldl, but in the reverse  
order of L

**head(L)** – the element L[0]

**tail(L)** – new list:  
all elements of L but the head

**indices(L)** – new set:  
0.to(length(L) - 1)

## Boolean expressions

`p == q` – `p` equals `q`  
`not(b)` – Boolean “not”  
`p != q` – `not(p == q)`  
`p and q` – Boolean “and”  
`p or q` – Boolean “or”  
`p implies q` – `not(p) or q`  
`p iff q` – `p == q`  
`and { p1, ..., pk }`  
`p1 and ... and pk`  
`or { p1, ..., pk }`  
`p1 or ... or pk`

## Integer expressions

no overflows, priority top-to-bottom

`i^j` – `i` to the power of `j`  
`-i` – negation  
`i * j` `i / j` `i % j`  
`i + j` `i - j`  
`i < j` `i <= j` `i > j` `i >= j`

## Modules

```
module A {  
  // pure definitions  
  // constants  
  // state variables  
  // actions  
  // runs  
  // temporal operators  
}
```

```
module B {  
  // make all names of A visible in B  
  import A.*  
}
```

## Control flow

`if (p) e1 else e2` – `e1` if `p` is true, and `e2` otherwise

## Pure definitions

may be nested

`pure val N = 3 + 4` – bind a constant expression to `N`

`pure def max(i, j) = {  
 if (i > j) i else j  
}` – bind the operator over constants to `max`

`(x, y) => max(i, j)` – an anonymous operator (lambda). Pass to other operators.

## States and definitions

`const Nodes: Set[str]` – declare a specification parameter, bind later with instance

`var active: Set[str]` – declare a state variable, uninitialized

`val allActive =  
 active == Nodes` – define a constant in the current state

`def isActive(n) = {  
 n.in(active)  
}` – define an operator of `n` and of the current state

## Actions – to make state transitions

`active' = Nodes` – record that `active` must be set to `Nodes` in a next machine state. Return true.

`nondet n = oneOf(Nodes)`

`A` – pick an arbitrary element of `Nodes`, bind to `n`, call action `A`

`assert(active != Set())` – report error if condition is false

`action activate(n) = {  
 active' = active.union(Set(n))  
}` – define an action

`all {  
 isActive("a"),  
 activate("b"),  
}` – execute all actions in arbitrary order. Only if all actions return true, record the updates to the next state and return true. Otherwise, return false.

`any {  
 activate("a"),  
 activate("b"),  
}` – execute some action that returns true, record its updates to the next state, return true. If no such action is available, return false.

## Runs – tests and execution examples

`init.then(step)` – execute `init`. On true, update the state variables, execute `step`. On false, return false.

`step.repeated(n)` – execute `step` `n` times, in sequence. Return true, only if all actions returned true.

`step.fail()` – execute `step`. If it returns false, return true. If it returns true, return false.

```
run test1 =  
  activate("a")  
  .then(activate("b"))  
  .then(all {  
    assert("a".in(active)),  
    assert("b".in(active)),  
    active' = active,  
  }) – a simple test
```

## Temporal operators

safety and liveness



under construction



under construction