Project Fortress Quick Vocabulary Guide

	ASCII	Rendered	Explanation
basic functions	`run()`	*run*()	application entry point
	`print(a:Any), println(a:Any)`	*print*(a: Any) , *println*(a: Any)	print anything, print anything with newline
		assert(x: Boolean)	fails if x is false
		assert(x: Boolean, *message*: String)	fails if x is false
		deny(x: Boolean, *message*: String)	fails if x is true
	`fail[\T\](s:String)`	*fail*$[\![T]\!]$(s: String)	print error message and throw the exception FailCalled
		getEnvironment(k: String, v: String)	get `java.property` or `ENVIRONMENT_VARIABLE`, returns v if neither is set
		CaseInsensitiveString(s: String)	strings with case-insensitive comparison
juxtaposition	`println "Hello world"`	*println* "`Hello world`"	function application
	`"Hello"  " "  "world"`	"`Hello`" " " "`world`"	string concatenation
	`3 5 = 15, 3 DOT 5 = 15`	$3\,5 = 15$, $3 \cdot 5 = 15$	multiplication (may also use `DOT`)
`Char`	`'c', '+', 'UNION', 'theta'`	'`c`' , '`+`' , '$\cup$' , 'θ'	character
`String`	`"cat" ||| "dog"`	"`cat`" $\|\|\|$ "`dog`"	string concatenation with whitespace unless empty
	`S1 // S2`	S1 $/\!/$ S2	string concatenation with newline
	`S1 || S2`	S1 $\|\|$ S2	string concatenation
	`strToInt("43")`	*strToInt*("43")	convert a string to $\mathbb{Z}32$
	`"I see " (3 + 2) " cats"`	"`I see` " $(3+2)$ " `cats`"	string concatenation does automatic *asString*
`FileReadStream`	`f = FileReadStream("/dev/stdin")`	$f =$ FileReadStream("`/dev/stdin`")	constructor
	`for l <- f.getLines() do ...`	`for` $l \leftarrow f.$*getLines*() `do` ...	generates lines (strings)
	`f.close()`	$f.$*close*()	
`Any`	`x === y`	$x \equiv y$	strict equivalence (pointer equality)
`Equality`	`=, =/=`	$=, \neq$	equals, not equals (numerical or value equality)
`PartialOrder`, `TotalOrder`	`<=, >=, <, >`	$\leq, \geq, <, >$	comparisons
`Comparison`, `LessThan` `GreaterThan`, `EqualTo`	`x CMP y`	x `CMP` y	returns LessThan, EqualTo, GreaterThan (or Unordered, if working with a PartialOrder)
numbers	`ZZ32, ZZ64`	$\mathbb{Z}32$, $\mathbb{Z}64$	32-bit signed integer, 64-bit signed integer
	`NN32, NN64`	$\mathbb{N}32$, $\mathbb{N}64$	32-bit unsigned integer, 64-bit unsigned integer
	`RR32, RR64`	$\mathbb{R}32$, $\mathbb{R}64$	32-bit floating point, 64-bit floating point
	`ZZ, QQ`	$\mathbb{Z}$, $\mathbb{Q}$	arbitrary precision integers and rationals
	`|x|`	$\|x\|$	absolute value, magnitude of number
	`2+3, 5-1, 3 DOT 5, 4/7`	$2+3$, $5-1$, $3 \cdot 5$, $4/7$	arithmetic on numbers
	`SUM[x <- 1:10] x`	$\sum_{x \leftarrow 1:10} x$	summation (use `PROD` for product $\prod$)
	`random(x)`	*random*(x)	$\mathbb{R}64$ value chosen randomly from $[0, x)$
	`|/x|, |\x/|`	$\lceil x \rceil$, $\lfloor x \rfloor$	ceiling, floor
	`x MIN y, x MAX y`	x `MIN` y , x `MAX` y	choose smaller value, choose larger value
	`BIG MIN[x <- 1:10] x`	$\mathtt{MIN}_{x \leftarrow 1:10} x$	`MIN` reduction (similarly for `MAX`)
	`SQRT x, x^y`	$\sqrt{x}$, x^y	square root, exponentiation

	ASCII	Rendered	Explanation
Boolean	`NOT e, a OPLUS b, a EQUIV b, a IMPLIES b`	$\neg e$, $a \oplus b$, $a \equiv b$, $a \to b$	inverse, exclusive OR, equivalence, implication
	`a AND b, a OR b`	$a \wedge b$, $a \vee b$	evaluates both arguments (AND and OR)
	`a AND: b, a OR: b`	$a \wedge\!: b$, $a \vee\!: b$	shortcut evaluation (AND and OR)
	`BIG AND[x<-a] f(x), BIG OR[x<-a] f(x)`	$\bigwedge_{x \leftarrow a} f(x)$, $\bigvee_{x \leftarrow a} f(x)$	reduction with $\wedge$ and $\vee$ (may also use $\oplus$ and $\equiv$)
Maybe	`x.getDefault(v)`	$x.getDefault(v)$	gets the value from x if present, otherwise returns v
Just	`Just(v)`	$\text{Just}(v)$	present value v
Nothing	`Nothing, Nothing[\ZZ32\]`	Nothing, $\text{Nothing}[\![\mathbb{Z}32]\!]$	missing value
Generator	`seq(1:30)`	$seq(1:30)$	sequential (SLOW) version of a generator
ranges	`start#count`	$start\#count$	counted range (3#6 is 3, 4, 5, 6, 7, 8)
	`start:till`	$start:till$	bounded range (3:6 is 3, 4, 5, 6)
	`start:till:step`	$start:till:step$	strided bounded range (3:7:2 is 3, 5, 7)
subscripting	`x[3], x[1:4], x[3,5], x[ "key" ]`	x_3 , $x_{1:4}$, $x_{3,5}$, $x[\text{“key”}]$	indexed selection from an array, list, or map
arrays	`array1[\T,Size\](initial)`	$array_1[\![T, \text{Size}]\!](initial)$	1-D array constructor
	`array1[\T,Size\](f:ZZ32->T)`	$array_1[\![T, \text{Size}]\!](f\!:\mathbb{Z}32 \to T)$	1-D array constructor with index-dependent initializer
	`array2[\T,S1,S2\](initial)`	$array_2[\![T, \text{S1}, \text{S2}]\!](initial)$	2-D array constructor
	`array2[\T,S1,S2\](f:(ZZ32,ZZ32)->T)`	$array_2[\![T, \text{S1}, \text{S2}]\!](f\!:(\mathbb{Z}32, \mathbb{Z}32) \to T)$	2-D array constructor wih index-dependent initializer
Indexed		`for` $i \leftarrow a.indices$ `do ... end`	generator for array indices
		`for` $(i,v) \leftarrow a.indexValuePairs$ `do ... end`	generator for pairs of index and corresponding value
collections	`|c|`	$\|c\|$	size of a string, list, set, map, ...
	`x IN c`	$x \in c$	membership in collection
	`c.isEmpty`	$c.isEmpty$	emptiness of collection
List	`<|[\T\]|>`	$\langle [\![T]\!] \rangle$	empty list (type required)
	`<|x, y, z|>`	$\langle x, y, z \rangle$	list aggregate
	`<|[\T\] x, y, z|>`	$\langle [\![T]\!]\, x, y, z \rangle$	list aggregate with type specified
	`<|[\T\] f(x) | x <- a, x >= 10|>`	$\langle [\![T]\!]\, f(x) \mid x \leftarrow a, x \geq 10 \rangle$	list comprehension
	`<|x, y|>.addRight(z)`	$\langle x, y \rangle.addRight(z)$	same as $\langle x, y \rangle \parallel \langle z \rangle$ (similarly $addLeft$)
	`S1 || S2`	S1 $\parallel$ S2	list concatenation
Set	`{[\T\]}`	$\{[\![T]\!]\}$	empty set (type required)
	`{x, y, z}`	$\{x, y, z\}$	set aggregate
	`{[\ZZ32\] x^2+y^2 | x<-1:10, y<-1:10}`	$\{[\![\mathbb{Z}32]\!]\, x^2 + y^2 \mid x \leftarrow 1\!:\!10, y \leftarrow 1\!:\!10\}$	set comprehension
Map	`{[\K,V\]}`	$\{[\![K, V]\!]\}$	empty map (types required)
	`{w |-> x, y |-> z}`	$\{w \mapsto x, y \mapsto z\}$	map aggregate
	`{[\ZZ,QQ\] x |-> 1/x | x<-1:10 }`	$\{[\![\mathbb{Z}, \mathbb{Q}]\!]\, x \mapsto 1/x \mid x \leftarrow 1\!:\!10\}$	map comprehension
		$map.add(key, val)\!:\!\text{Map}$	returns new map with $key \mapsto val$ mapping updated
		$map.union(f\!:(k, v_1, v_2) \to v, other\!:\!\text{Map})\!:\!\text{Map}$	combines two maps, calling f for new value when key is present in both
		$map.member(k\!:K)\!:\!\text{Maybe}[\![V]\!]$	returns value for k if k is in map, else returns Nothing
		$map.member(k\!:K, v\!:V)\!:V$	returns value for k if k is in map, else returns v
		$map.extractMaximum()\!:\!\text{Maybe}[\![(K, V, \text{Map}[\![K, V]\!])]\!]$	if non-empty, returns maximum key, its value, and new map with key removed