SubScript Cheat Sheet

by anatoliykmetyuk via cheatography.com/25536/cs/6718/

Syntax

`import subscript.language` `import subscript.Predef._`	Top-level imports required in all SubScript sources.
`script a = expr`	Script definition

`script..`
`  a = expr`
`  b = expr`
	Shorthand script definition

`runScript(script _name)`	Run scripts like this
`[ expr ]`	Prioritizing Parentheses (like "-()" in "2 - (1 + 3)", just for scripts)
`[** expr **]`	Launch Anchor
`[* expr *]`	Launch
`@a: b`	Annotation
`@{prin tln (th ere)}: a`	Also annotation. There points to the annotated expression node
`var x: Int = 3`	Variable declarations are possible in scripts
`let scala_expr`	Executes `scala_expr` as a tiny code fragment.

Sequential Operators

`a ; b`	Executes next operator as soon as current one has success
`a b`	Same as above
`a` `b`	Same as above

Parallel Operators

`a && b`	**Non-strict and-parallelism**. Succeeds iff all its operands do. On failure of one of the children terminates without success immediately.
`a & b`	**Strict and-parallelism**. Same as above, but if some of its children doesn't have success, it waits for the rest of the children to execute before terminating.
`a || b`	**Non-strict or-parallelism**. Succeeds iff at least one of its children does. After a children succeeds, it terminates immediately with success.

Parallel Operators (cont)

`a | b`	**Strict or-parallelism**. Same as above, but waits for the rest of the children after one succeeds. Has success immediately after at least one child succeeds (termination and success are not the same things).

Result Values

`runScript(script_name).$`	From Scala code, returns the result value of `script _name` script, as `Try[Any]`.
`a^`	From SubScript code, sets the result of the parent script to that of `a`. E.g. in `script foo = a^ b c`, script `foo` will have a result of `a`. `b` and `c` are still executed as usually.
`a^^`	The result of the parent script becomes a `Seq[Any]`. The result of `a` is recorded into that `Seq` at the index equal to `a`'s current pass (that is, first pass in a loop will go to index 0, second - to 1 etc).
`a^^int_li teral`	The result of the parent script becomes a tuple. `a`'s result is recorded at `int_li teral`-th position to the tuple. E.g. `a^^1 b^^2` will result in a tuple with `_1` set to `a`'s result and `_2` - to `b`'s result.
`^literal`	Sets the result of the parent script to `literal`. E.g. `^5`, `^"Foo"`, `^'x'`.
`^literal^^`	Sets the result to `Seq[Any]`, records `literal` under its `pass`'s index.
`^literal^^int_li teral`	Sets the result to a tuple, places this `literal` under `int_li teral`-th position in this tuple.

Scala Code Blocks

`{! scala block !}`	**Normal** code block. Activation, Execution, Deactivation.
`{: scala block :}`	**Tiny** code block. Execution on Activation.
`{. scala block .}`	**Event-handling** code block. Does not execute automatically, need manual execution.
`{* scala block *}`	**Threaded** code block. Executes from a new thread (all the other blocks execute from Script Executor's thread).

Special Operands

`[+]`	**Epsilon**, or empty action. Has success immediately after activation.
`[-]`	**Delata**, or deadlock. Terminates without success immediately after activation.
`...`	**Loop**. When used as an operand to a sequence, loops the sequence. E.g. `a b ...` executes in order "a b a b a b" etc as an infinite loop. `a ... b` and `... a b` have same effect.
`break`	**Break**. Breaks activation of its parent operator.
`break?`	**Optional break**. Behaves like `break`, but resumes activation after an action happened in an operand activated before itself.
`..?`	**Optional break loop**. Mixes together `break?` and `...`.

Alternative Operators

`a + b`	**Choice**. Starts with `a` and `b` activated. When either starts executing, excludes another.
`a / b`	**Disruption**. Executes `a` until `b` starts, then excludes (terminates) `a` and continues with `b`. If `a` gets terminated without `b` ever getting started, excludes `b`.

Conditional Operators

`if scala_expr then expr else expr`	Executes `then` part if `scala_expr` is `true`, otherwise - `else` part.
`do expr then expr else expr`	Executes `do` part first. If it has success, executes `then` part, otherwise - `else` part.

Dataflow

`a ~~(x: T)~~> b`	**Dataflow**. Executes `a`, casts its result to type `T`, assigns it to `x` and executes `b` with `x` in scope.
`a ~~(x: T)~~> b` `+~/~(x: E)~~> c`	Dataflow with an extra clause to handle exceptions. If `a` succeeds, the behaviour is as in the case above. Otherwise, an exception with which `a` failed is casted to `E` (which must be `<: Throwable`) and handled by `c`. Like `catch` in `try-catch`.

`a ~~(x: T)~~> b`
`+~~(y: A)~~> c`
`+~~(z: B)~~> d`
	Dataflow can arbitrary number of result-handling clauses and exception-handling clauses.

`a ~~(x: T)~~^ scala_expr` `+~~(x: A)~~^ scala_expr`	**Dataflow map**. Similar to Dataflow, but runs the result of `a` through a given `scala_expr` and sets the result of it as the result of the parent script.
`a ~~^ f`	Shorthand for `a ~~(x: T)~~^ f(x)`.