

Specware® 4.0.5 Quick Reference

Processing Commands

:sw-help	Print list of processing commands
:swpath <i>path</i> ;...; <i>path</i>	Set SWPATH environment variable
:swpath	Print SWPATH
:dir	List files in current folder
:cd <i>folder-name</i>	Change current folder
:sw [<i>unit-id</i>]	Process unit(s)
:show [<i>unit-id</i>]	Process and print unit
:list	List current units in cache
:sw-init	Clear unit cache
:swl <i>spec-unit-id</i> [<i>target-file</i>]	Generate Lisp from spec
:cl <i>lisp-file</i>	Load Lisp file
:swll <i>spec-unit-id</i>	Incrementally generate and load Lisp
:swe-spec <i>spec-unit-id</i>	Set context for :swe command
:swe <i>expr</i>	Evaluate and print Metaslang expression

Units (specs, morphisms, diagrams, ...)

[[/]<i>name</i>/.../<i>name</i>][#<i>name</i>]	Unit-identifier
<i>unit-id</i> = <i>unit-term</i>	Unit-definition
<i>spec</i> <i>declaration</i> ... <i>endspec</i>	Returns spec-form
<i>qualifier</i> <i>qualifying spec</i>	Qualifies unqualified sort- and op-names
<i>translate spec by</i> {[<i>sort</i> <i>op</i>] <i>name</i> +-> <i>name</i>, ...}	Spec-translation: replaces lhs names in spec by rhs names
<i>spec</i> [<i>morphism</i>]	Spec-substitution: replaces source spec of morphism by target spec in the given spec
<i>colimit diagram</i>	Returns spec at apex of colimit cocone
<i>obligations spec-or-morphism</i>	Returns spec containing proof obligations
<i>morphism spec</i> -> <i>spec</i> {[<i>sort</i> <i>op</i>] <i>name</i> +-> <i>name</i>, ...}	Returns spec-morphism
<i>diagram</i> {<i>diagram-node-or-edge</i>, ...}	Returns diagram
<i>name</i> +-> <i>spec</i>	Diagram-node
<i>name</i> : <i>name</i> -> <i>name</i> +-> <i>morphism</i>	Diagram-edge
<i>generate lisp spec</i> [<i>in</i> "<i>filename</i>"]	Generates Lisp code
<i>prove claim in spec</i> [<i>with snark</i>] [<i>using</i> {<i>claim</i>, ...}] [<i>options prover-options</i>]	Proof-term

Names

[<i>qualifier.</i>] <i>name</i>	Sort-name, op-name
<i>word-symbol</i>	Qualifier
<i>word-symbol</i> <i>non-word-symbol</i>	Name, constructor, field-name, (sort-)var
A3 posNat? z_k	Examples of word-symbols
~! @\$^ &*~ =+ \ :< >/?	Examples of non-word-symbols

Literals

true false	Boolean-literal
0 1 ...	Nat-literal
#<i>char-glyph</i> #"	Char-literal
" <i>char-glyph</i>..."	String-literal
A ... Z a ... z 0 ... 9 ! : # ... \\ \" \a \b \t \n \v \f \r \s \x00 ... \xff	Char-glyph

Declarations and Definitions

import <i>spec</i>	Import-declaration
sort <i>sort-name</i>	Sort-declaration
sort <i>sort-name</i> <i>sort-var</i> sort <i>sort-name</i> (<i>sort-var</i> , ...)	Polymorphic sort-declaration
sort <i>sort-name</i> [<i>sort-vars</i>] = <i>sort</i>	Sort-definition
op <i>op-name</i> [infixl infixr <i>prio</i>] : [fa (<i>sort-var</i> , ...)] <i>sort</i>	Op-declaration; optional infix assoc/prio; optional polymorphic sort parameters
def [fa (<i>sort-var</i> , ...)] <i>op-name</i> [<i>pattern</i> ...] = <i>expr</i>	Op-definition; optional polymorphic sort parameters; optional formal parameters
axiom theorem conjecture <i>name</i> = [sort fa (<i>sort-var</i> , ...)] <i>expr</i>	Claim-definition; optional polymorphic sort parameters

Sorts

<i>constructor</i> [<i>sort</i>] ... <i>constructor</i> [<i>sort</i>]	Sum sort
<i>sort</i> -> <i>sort</i>	Function sort
<i>sort</i> * ... * <i>sort</i>	Product sort
{ <i>field-name</i> : <i>sort</i> , ...}	Record sort
(<i>sort</i> <i>expr</i>)	Subsort (Sort-restriction)
{ <i>pattern</i> : <i>sort</i> <i>expr</i> }	Subsort (Sort-comprehension)
<i>sort</i> / <i>expr</i>	Quotient sort
<i>sort</i> <i>sort</i> ₁ <i>sort</i> (<i>sort</i> ₁ , ...)	Sort-instantiation

Expressions

fn [] <i>pattern</i> -> <i>expr</i> ...	Lambda-form
case <i>expr</i> of [] <i>pattern</i> -> <i>expr</i> ...	Case-expression
let <i>pattern</i> = <i>expr</i> in <i>expr</i> let <i>rec-let-binding</i> ... in <i>expr</i>	Let-expression
def <i>name</i> [<i>pattern</i> ...][[: <i>sort</i>] = <i>expr</i>	Rec-let-binding; optional formal parameters
if <i>expr</i> then <i>expr</i> else <i>expr</i>	If-expression
fa ex (<i>var</i> , ...) <i>expr</i>	Quantification (non-constructive)
<i>expr</i> <i>expr</i> ₁ ... <i>expr</i> ₁ <i>op-name</i> <i>expr</i> ₂	Application (prefix- or infix-application)
<i>expr</i> : <i>sort</i>	Annotated-expression
<i>expr</i> . <i>N</i> <i>expr</i> . <i>field-name</i>	Field-selection, product sort (<i>N</i> = 1 2 3 ...) Field-selection, record sort
(<i>expr</i> , <i>expr</i> , ...)	Tuple-display (has product sort)
{ <i>field-name</i> = <i>expr</i> , ...}	Record-display (has record sort)
[<i>expr</i> , ...]	List-display
project relax restrict quotient choose <i>expr</i>	Various structors
[embed] <i>constructor</i>	Embedder
embed? <i>constructor</i>	Embedding-test
<i>op-name</i>	Op-name
<i>var</i>	Local-variable
<i>literal</i>	Literal

Patterns

<i>pattern</i> : <i>sort</i>	Annotated-pattern
var as <i>pattern</i>	Aliased-pattern
<i>pattern</i> _{hd} : : <i>pattern</i> _{tl}	Cons-pattern
<i>constructor</i> [<i>pattern</i>]	Embed-pattern
(<i>pattern</i> , <i>pattern</i> , ...)	Tuple-pattern
{ <i>field-name</i> = <i>pattern</i> , ...}	Record-pattern
[<i>pattern</i> , ...]	List-pattern
quotient <i>expr</i> <i>pattern</i>	Quotient-pattern
relax <i>expr</i> <i>pattern</i>	Relax-pattern
—	Wildcard-pattern
<i>var</i>	Variable-pattern
<i>literal</i>	Literal-pattern