Specware® 4.0.5 Quick Reference

Processing Commands

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

Units (specs, morphisms, diagrams, …)

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

Names

[*qualifier***.**] *name*	Sort-name, op-name
word-symbol	Qualifier
word-symbol | *non-word-symbol*	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
#char-glyph | **#"**	Char-literal
" *char-glyph*… **"**	String-literal
A | … | **Z** | **a** | … | **z** | **0** | … | **9** | **!** | **:** | **#** | … | **** | **\"** | **\a** | **\b** | **\t** | **\n** | **\v** | **\f** | **\r** | **\s** | **\x00** | … | **\xff**	Char-glyph

Declarations and Definitions

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

Sorts

| *constructor* [*sort*] **|** … **|** *constructor* [*sort*]	Sum sort
sort **->** *sort*	Function sort
sort ***** … ***** *sort*	Product sort
{*field-name* **:** *sort***,** …**}**	Record sort
(*sort* **|** *expr***)** **{***pattern* **:** *sort* **|** *expr***}**	Subsort (Sort-restriction) Subsort (Sort-comprehension)
sort **/** *expr*	Quotient sort
sort $sort_1$ *sort***(***$sort_1$***,** …**)**	Sort-instantiation

Expressions

fn [**|**] *pattern* **->** *expr* **|** …	Lambda-form
case *expr* **of** [**|**] *pattern* **->** *expr* **|** …	Case-expression
let *pattern* **=** *expr* **in** *expr* **let** *rec-let-binding* … **in** *expr*	Let-expression
def *name* [*pattern* …][**:** *sort*] **=** *expr*	Rec-let-binding; optional formal parameters
if *expr* **then** *expr* **else** *expr*	If-expression
fa|**ex(***var***,** …**)** *expr*	Quantification (non-constructive)
expr $expr_1$ … | $expr_1$ *op-name* $expr_2$	Application (prefix- or infix-application)
expr **:** *sort*	Annotated-expression
expr **.** *N* *expr* **.** *field-name*	Field-selection, product sort (*N* = **1**|**2**|**3**|…) Field-selection, record sort
(*expr***,** *expr***,** …**)**	Tuple-display (has product sort)
{*field-name* **=** *expr***,** …**}**	Record-display (has record sort)
[*expr***,** …**]**	List-display
project|**relax**|**restrict**|**quotient**|**choose** *expr*	Various structors
[**embed**] *constructor*	Embedder
embed? *constructor*	Embedding-test
op-name	Op-name
var	Local-variable
literal	Literal

Patterns

pattern **:** *sort*	Annotated-pattern
var **as** *pattern*	Aliased-pattern
$pattern_{hd}$ **::** $pattern_{tl}$	Cons-pattern
constructor [*pattern*]	Embed-pattern
(*pattern***,** *pattern***,** …**)**	Tuple-pattern
{*field-name* **=** *pattern***,** …**}**	Record-pattern
[*pattern***,** …**]**	List-pattern
quotient *expr pattern*	Quotient-pattern
relax *expr pattern*	Relax-pattern
_	Wildcard-pattern
var	Variable-pattern
literal	Literal-pattern