Specware® 4.1 Quick Reference

Shell Commands

help [*command*]	Print help for shell commands
cd [*folder-name*]	Change or print current folder
dir | **dirr**	List `.sw` files in folder (current or recursively)
path [*path*`;`…`;`*path*]	Set or print SWPATH environment variable
p[**roc**] [*unit*]	Process unit(s)
cinit	Clear unit cache
show | **showx** [*unit*]	Process and print unit (normal or extended form)
punits | **lpunits** [*unit* [*target-file*]]	Generate proof-units for unit (global or local)
ctext [*spec*]	Sets context for evaluation
e[**val**] | **eval-lisp** [*expression*]	Evaluate and print expression (directly or in Lisp)
gen-lisp | **lgen-lisp** [*spec* [*target-file*]]	Generate Lisp from spec (global or local)
gen-java [*spec* [*options-spec*]]	Generate Java from spec
gen-c [*spec* [*target-file*]]	Generate C from spec
make [*spec*]	Generate C with makefile and call "make" on it
ld | **cf** | **cl** [*lisp-file*]	Load, compile, or load+compile Lisp file
exit | **quit**	Terminate shell

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 type- and op-names
translate *spec* **by** **{**[**type** | **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* **{**[**type** | **op**] *name* **+->** *name***,** …**}**	Returns spec-morphism
diagram {*diagram-node-or-edge***,** …**}**	Returns diagram
name **+->** *spec*	Diagram-node
name **:** *name* **->** *name* **+->** *morphism*	Diagram-edge
generate [**c** | **java** | **lisp**] *spec* [**in "***filename***"** | **with** *options-spec*]	Generates C, Java, or Lisp code
prove *claim* **in** *spec* [**with snark**] [**using {***claim***,** …**}**] [**options** *prover-options*]	Proof-term

Names

[*qualifier***.**] *name*	Type-name, op-name
word-symbol	Qualifier
word-symbol | *non-word-symbol*	Name, constructor, field-name, (type-)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
type *type-name*	Type-declaration
type *type-name type-var* **type** *type-name* **(***type-var***,** …**)**	Polymorphic type-declaration
type *type-name* [*type-var* | **(***type-vars***)**] **=** *type*	Type-definition
op *op-name* [**infixl** | **infixr** *prio*] **:** [**[***type-var***,** …**]**] *type*	Op-declaration; optional infix assoc/prio; optional polymorphic type parameters
def [**[***type-var***,** …**]**] *op-name* [*pattern* …] [**:** *type*] **=** *expr*	Op-definition; optional polymorphic type parameters; optional formal parameters
axiom|**theorem**|**conjecture** *name* **is** [**[***type-var***,** …**]**] *expr*	Claim-definition; optional polymorphic type parameters

Types

| *constructor* [*type*] **|** … **|** *constructor* [*type*]	Sum type
type **->** *type*	Function type
type ***** … ***** *type*	Product type
{*field-name* **:** *type***,** …**}**	Record type
(*type* **|** *expr***)** **{***pattern* **:** *type* **|** *expr***}**	Subtype (Type-restriction) Subtype (Type-comprehension)
type **/** *expr*	Quotient type
type type$_1$ *type***(***type*$_1$**,** …**)**	Type-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* …][**:** *type*] **=** *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)
restrict *expr expr*$_1$	Restrict-expression
expr **:** *type*	Annotated-expression
expr **.** *N* *expr* **.** *field-name*	Field-selection, product type (*N* = **1**|**2**|**3**|…) Field-selection, record type
(*expr***,** *expr***,** …**)**	Tuple-display (has product type)
{*field-name* **=** *expr***,** …**}**	Record-display (has record type)
[*expr***,** …**]**	List-display
project|**relax**|**quotient**|**choose** *expr*	Various structors
[**embed**] *constructor*	Embedder
embed? *constructor*	Embedding-test
op-name	Op-name
var	Local-variable
literal	Literal

Patterns

pattern **:** *type*	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