

---

# Specware Quick Reference Documentation

*Release 4.2*

**Kestrel Institute**

March 25, 2013

## Contents

<b>1</b>	<b>Shell Commands</b>	<b>ii</b>
<b>2</b>	<b>Units (specs, morphisms, diagrams, ...)</b>	<b>ii</b>
<b>3</b>	<b>Names</b>	<b>iii</b>
<b>4</b>	<b>Declarations and Definitions</b>	<b>iii</b>
<b>5</b>	<b>Types</b>	<b>iii</b>
<b>6</b>	<b>Expressions</b>	<b>iv</b>
<b>7</b>	<b>Patterns</b>	<b>iv</b>

---

# 1 Shell Commands

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

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

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

### 3 Names

Syntax	Construct
<i>[qualifier.] name</i>	Type-name, op-name
<i>word-symbol</i>	Qualifier
<i>word-symbol</i>   <i>non-word-symbol</i>	Name, constructor, field-name, (type-)var
A3   posNat?   z-k	Examples of word-symbols
`~!   @\$^   &* -   =+ \     : <   > / ?	Examples of non-word-symbols
true   false	Bool-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

### 4 Declarations and Definitions

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

### 5 Types

Syntax	Construct
`` *constructor* [ *type* ] ``   ...   <i>constructor</i> [ <i>type</i> ]	Sum type
<i>type</i> -> <i>type</i>	Function type
<i>type</i> * ... * <i>type</i>	Product type
{ <i>field-name</i> : <i>type</i> , ... }	Record type
( <i>type</i>   <i>expr</i> )	Subtype (Type-restriction)
{ <i>pattern</i> : <i>type</i>   <i>expr</i> }	Subtype (Type-comprehension)
<i>type</i> / <i>expr</i>	Quotient type
<i>type type1 type(type1, ...)</i>	Type-instantiation

## 6 Expressions

<b>fn</b> <i>[] pattern -&gt; expr</i>   ...	<b>Lambda-form</b>
<b>case</b> <i>expr of [] pattern -&gt; expr</i>   ...	Case-expression
<i>let</i> <i>pattern = expr in expr</i>	Let-expression
<b>let</b> <i>rec-let-binding ... in expr</i>	
<b>def</b> <i>name [pattern ...][: type ] = expr</i>	Rec-let-binding; optional formal parameters
<b>if</b> <i>expr then expr else expr</i>	If-expression
<b>fa</b>   <b>ex</b> <i>(var, ...) expr</i>	Quantification (non-constructive)
<i>expr expr1 ...</i>   <i>expr1 op-name expr2</i>	Application (prefix- or infix-application)
<i>expr : type</i>	Annotated-expression
<i>expr . N</i>	Field-selection, product type (N = 1 2 3  ...)
<i>expr . field-name</i>	Field-selection, record type
<i>(expr, expr, ...)</i>	Tuple-display (has product type)
<i>{ field-name = expr, ... }</i>	Record-display (has record type)
<i>[ expr, ... ]</i>	List-display
<b>project</b>   <b>quotient</b>   <b>choose</b> <i>expr</i>	Various structors
<b>[embed]</b> <i>constructor</i>	Embedder
<b>embed?</b> <i>constructor</i>	Embedding-test

## 7 Patterns

Syntax	Construct
<i>pattern : type</i>	Annotated-pattern
<i>var as pattern</i>	Aliased-pattern
<i>patternhd :: patternl</i>	Cons-pattern
<i>constructor [pattern]</i>	Embed-pattern
<i>( pattern , pattern, ... )</i>	Tuple-pattern
<i>{ field-name = pattern , ... }</i>	Record-pattern
<i>[ pattern , ... ]</i>	List-pattern
<i>pattern   expr</i>	Guarded-pattern
<i>_</i>	Wildcard-pattern
<i>var</i>	Variable-pattern
<i>literal</i>	Literal-pattern