

---

# **Specware User Manual**

***Release 4.2***

**Kestrel Institute**

May 21, 2013



# CONTENTS

<b>1</b>	<b>Installing Specware</b>	<b>1</b>
1.1	Contents of Distribution Package . . . . .	1
1.2	System Requirements . . . . .	1
1.3	Installation Instructions . . . . .	2
1.4	Uninstalling . . . . .	2
<b>2</b>	<b>Getting Started</b>	<b>3</b>
2.1	Starting Specware . . . . .	3
2.2	Exiting Specware . . . . .	3
<b>3</b>	<b>Usage Model</b>	<b>5</b>
3.1	Units . . . . .	5
3.2	Interaction . . . . .	5
<b>4</b>	<b>Defining Units</b>	<b>7</b>
4.1	Conceptual Model . . . . .	7
4.2	Realization via the File System . . . . .	7
4.3	Unit Definitions Are Managed Outside of Specware . . . . .	9
<b>5</b>	<b>The Specware Shell</b>	<b>11</b>
5.1	Overview . . . . .	11
5.2	Typechecking Specs . . . . .	12
5.3	Resolution of Unit Identifiers . . . . .	12
5.4	Command Format . . . . .	15
5.5	Miscellaneous Commands . . . . .	15
5.6	Processing a Unit . . . . .	16
5.7	Showing a Unit . . . . .	17
5.8	Generating Proof Units . . . . .	17
5.9	Evaluating Expressions . . . . .	18
5.10	Generating Lisp Code . . . . .	19
5.11	Auxiliary Commands for Lisp . . . . .	28
5.12	Finally . . . . .	28
<b>6</b>	<b>Proving Properties in Specs</b>	<b>29</b>
6.1	The Proof Unit . . . . .	29
6.2	Implicit Axioms . . . . .	30
6.3	Proof Errors . . . . .	30
6.4	Proof Log Files . . . . .	30
6.5	Multiple Proofs . . . . .	30
6.6	Interrupting Snark . . . . .	30

6.7	The Prover Base Library . . . . .	31
6.8	The Experimental Nature of the Prover . . . . .	31
<b>7</b>	<b>Lisp Code Generated from Specs</b>	<b>33</b>
7.1	Translation of Specware Names to Lisp Names . . . . .	33
7.2	Arity and Currying Normalization . . . . .	33
7.3	Representation of Other Types . . . . .	34
<b>8</b>	<b>Debugging Generated Lisp Files</b>	<b>35</b>
8.1	Tracing . . . . .	35
8.2	Breaking . . . . .	35
8.3	Timing . . . . .	35
8.4	Interrupting . . . . .	36
<b>9</b>	<b>Emacs Usage in Specware</b>	<b>37</b>
9.1	Specware Mode Commands . . . . .	37
9.2	Specware Interaction Commands . . . . .	38
9.3	Other Useful Emacs Commands . . . . .	39
9.4	X-Symbol Mode . . . . .	39
9.5	Hide/Show Commands . . . . .	39
	<b>Index</b>	<b>41</b>

# INSTALLING SPECWARE

## 1.1 Contents of Distribution Package

The following programs and documents are included on the Specware 4.2 installation CD:

**setup.exe** This program in the root directory of the CD is the Specware 4.2 installer for Windows. It should be launched automatically when the CD is inserted into the CD-ROM drive.

**XEmacs** This folder contains the XEmacs `setup.exe` installer for Windows, as well as the packages necessary to install the program. XEmacs is the environment within which Specware 4.2 is designed to run.

**SNARK** This folder contains the license agreement and modified source code for the theorem prover SNARK, which is built into Specware 4.2.

The following documentation is included with the distribution package:

**Specware 4.2 Quick Reference** The Quick Reference gives an overview of processing commands and Metaslang language constructs.

**Specware 4.2 User Manual (this document)** The User Manual serves as a quick guide to basic usage and concepts of Specware. After reading this, you should feel comfortable with the mechanics of running and using Specware.

**Specware 4.2 Tutorial** The Tutorial will guide you through the process of specifying, refining and generating code in Specware. A comprehensive example provides step-by-step instructions for this development process.

**Specware 4.2 Language Manual** The Language Manual discusses the Metaslang specification language and gives the grammar rules and meaning for each Metaslang language construct.

## 1.2 System Requirements

### 1.2.1 Hardware

Specware has relatively modest system requirements for simple projects. Of course, as with any development tool, as your projects being developed become more complex, you may wish to work on a more powerful machine. For average use, however, the following basic hardware configuration is recommended:

- CPU: 250 Mhz
- RAM: 128 MB total, at least 64 MB free for applications
- Disk space: 15 MB for base system, 10-50 MB for user projects

### **1.2.2 Operating system**

This version of Specware 4.2 has been tested to work with Windows XP, Windows 2000 and Windows NT 4.0.

## **1.3 Installation Instructions**

1. Insert the installation CD into your CD-ROM drive; the Specware 4.2 `setup.exe` installer wizard will be automatically launched.
2. After accepting the license agreement, the installer will try to find the path to the XEmacs `xemacs.exe` startup file under the `Program Files` directory. If the path is found, the installer continues. Otherwise, you will have the option to either install XEmacs from the distribution CD (click `Yes`) or to manually type in the full path to `xemacs.exe`, if it is installed elsewhere on your machine (click `No`). Clicking `Yes` will launch the XEmacs installer wizard. Select the “Install from Local Directory” option for quickest installation, specify `[CD-ROM drive] : \XEmacs\packages\` as the directory from which to install the packages, and click `Next` through the remaining steps in the wizard to accept the default configuration. After XEmacs has been installed, you will return to the Specware installer wizard.
3. Select the directory where you would like Specware to be installed (the default is `C:\Program Files\Specware4.2`), and click `Next` to complete the installation. A shortcut to Specware will be placed on your Desktop as well as in the Program Files folder in the Start menu. Documentation, libraries and example code will be placed in the installation directory you selected.

## **1.4 Uninstalling**

To uninstall Specware 4.2, run the installer on the CD again and select the “Remove” option in the wizard, or use the `Add/Remove Programs` setting in the Control Panel.

# GETTING STARTED

Specware is a development environment that runs on top of Lisp. The following sections describe the Specware environment and the basic mechanisms for running Specware.

## 2.1 Starting Specware

The user interacts with Specware through the “Specware Shell”, with the choice of running the latter in an XEmacs buffer or in a Lisp window. XEmacs users will almost certainly be more comfortable with the first choice, while the second choice may be preferable for users unfamiliar with Xemacs.

To start Specware, double-click either the *Specware4 XEmacs* shortcut or the *Specware4* shortcut on your Desktop, or select one of the two from the *Start Menu -> Program Files -> Specware* menu.

When Specware is launched through the *Specware4 XEmacs* shortcut, a couple of things happen: an Allegro Common Lisp Console window appears (which may be ignored but should not be closed), XEmacs is started, and a Lisp image containing the Specware Shell is started inside an XEmacs buffer entitled `Specware Shell`.

When Specware is launched through the *Specware4* shortcut, a Lisp window entitled `Specware Shell` appears in which the Specware Shell is running.

In either case, the Specware Shell starts with issuing a prompt (an asterisk “\*”), prompting the user to issue a Specware Shell command. All of the user interaction (see the next chapter) with Specware occurs at the Specware Shell prompt: the user issues a command; Specware processes it and shows any results or error messages, insofar as applicable; then prompts the user; and so on, until the end of the session.

Processing errors may cause the execution of the Specware Shell to be interrupted, throwing interaction into a Lisp shell. To return to the Specware Shell, issue the Lisp command `:sw-shell`. Alternatively, select an appropriate restart action if one is offered by Lisp.

## 2.2 Exiting Specware

To exit Specware, type **exit** or **quit** at the Specware prompt. This will terminate the Specware session.

An existing XEmacs window remains open and needs to be closed separately.





# USAGE MODEL

## 3.1 Units

Simply put, the functionality provided by Specware consists in the capability to construct specs, morphisms, diagrams, code, proofs, and other entities. All these entities are collectively called *units*.

Some of the operations made available by Specware to construct units are fairly sophisticated. Examples are colimits, extraction of proof obligations, discharging of proof obligations by means of external theorem provers, and code generation.

The Metaslang language is the vehicle to construct units. The language has syntax to express all the unit-constructing operations that Specware provides. The user defines units in Metaslang, writing the definitions in `.sw` files. (This file extension comes from the first letters of the two syllables in “Specware”.)

Currently, the only way to construct units in Specware is by writing text in Metaslang. The `.sw` files that define units are edited outside of Specware, using XEmacs, Notepad, Vim, or any other text editor of choice. These files are processed by Specware by giving suitable commands to the Specware Shell. Future versions of Specware will include the ability to construct units by other means. For instance, instead of listing the nodes and edges of a diagram in text, it will be possible to draw the diagram on the screen.

## 3.2 Interaction

The interaction between the user and Specware takes place through the Specware Shell.

When `.sw` files are processed by Specware, progress and error messages are displayed in the window containing the Specware Shell. In addition, the results of processing are saved into an internal cache that Specware maintains. Lastly, processing of certain kinds of units result in new files being created. For example, when Lisp code is generated from a spec, the code is deposited into a `.lisp` file.

From the Specware Shell it is possible to evaluate Metaslang expressions in the context of a given spec, either directly or through generated Lisp code.

Specware also features auxiliary commands to display information about units, inspect and clear the internal cache, and inspect and change the `SWPATH` environment variable, which determines how unit identifiers are resolved to `.sw` files.



# DEFINING UNITS

## 4.1 Conceptual Model

A unit definition consists of a unit identifier and a unit term. The identifier identifies the unit and the term defines how the unit is constructed.

A project developed with Specware consists of a set of unit definitions, some of which may come from libraries. Units have unique identifiers within the project.

### 4.1.1 Unit Identifiers

A unit identifier is a finite, non-empty sequence of word symbols (word symbols are defined in the Metaslang grammar). The sequence of word symbols is essentially a “path” in a tree: the units comprising a project are organized in a tree.

This provides a convenient and simple way to organize the units comprising a project. Libraries are subtrees of the whole tree. Parallel development of different parts of a project can be carried out in different subtrees that can be later put together without risk of naming clashes.

### 4.1.2 Unit Terms

A unit term is text written in the Metaslang language. Metaslang features various ways to construct specs, morphisms, and all the other kinds of units. For instance, it is possible to construct a spec by explicitly listing its types, ops, and axioms. It is also possible to construct a spec by applying the colimit operation to a diagram of specs and morphisms.

A unit term may reference other units. For instance, a spec constructed by extending another one references the spec being extended.

References can be “SWPATH-based” or “relative”. In either case they are resolved to full unit identifiers of units in the tree, according to simple rules explained later.

## 4.2 Realization via the File System

The conceptual model just described is realized by means of the file system of the underlying operating system. The file system has essentially a tree structure. The tree of units comprising a Specware project is mapped to subtrees of the file system; the word symbols comprising a path are mapped to file and directory names.

Future versions of Specware will have a more sophisticated UI that will realize the conceptual model directly. Users will graphically see the units organized in a tree and they will be able to add, remove, move, and edit them. The mapping to the file system may even be made totally transparent to the user.

### 4.2.1 The `SWPATH` Environment Variable

The mapping of the conceptual unit tree to the file system is defined by the environment variable `SWPATH`. Similarly to the `PATH` environment variable in operating systems, `SWPATH` is a string consisting of a semicolon-separated list of absolute directory paths in the file system.

Roughly speaking, the unit tree consists of all the units defined in `.sw` files under the directories listed in `SWPATH`. The identifier of each unit is its path from the directory in `SWPATH` under which the file defining the unit is: if the unit is under a directory named `ub2`, its identifier is its absolute path in the file system “minus” the `ub2` prefix. This approximate statement is made precise and illustrated by examples below.

### 4.2.2 Single Unit in a File

The simplest way to define a unit is to write its term into a `.sw` file in the subtree of one of the directories in `SWPATH`. The identifier of the unit is the name of the file, without `.sw`, prefixed by the path from the directory in `SWPATH` to the file.

For example, suppose that `SWPATH` includes the directory `C:\users\me\specware`. The user creates a file named `A.sw` immediately under the directory `C:\users\me\specware\one\two`, containing the following text:

```
spec
  type X
endspec
```

The absolute path of the file in the file system is `C:\users\me\specware\one\two\A.sw`. The unit is a spec declaring just a type `X`. The identifier of the unit is `/one/two/A`. Note that the path components are separated by “/” (forward slash), even though the underlying file system uses “\” (backward slash). Unit identifier are sequences of word symbols separated by “/”, regardless of the underlying operating system.

### 4.2.3 Multiple Units in a File

It is also possible to put multiple units inside a `.sw` file. The file must be in the subtree of one of the directories in `SWPATH`. Instead of just containing a unit term, the file contains one or more unit definitions. A unit definition consists of a word symbol, `=` (equal), and a unit term.

This case works almost exactly as if the file were replaced by a directory with the same name (without `.sw`) containing one `.sw` file for every unit defined therein. Each such file has the word symbol of the unit as name (plus `.sw`) and the term of the unit as content.

The only difference between the case of multiple units per file and the almost equivalent case where the file is replaced by a directory containing single-unit files, is that in the former case the last separator is not “/” but “#” (sharp). (This is reminiscent of the URI syntax, where subparts of a document are referred to using “#”.)

Suppose, as in the previous example, that `SWPATH` includes the directory `C:\users\me\specware`. The user creates a file named `three.sw` immediately under the directory `C:\users\me\specware\one\two`, containing the following text:

```
B = spec
  type Y
endspec
```

```
three = spec
    import B
    type Z
endspec
```

This file defines two specs, one declaring just a type `Y`, the other, next to importing the first spec, declaring just a type `Z`. The identifier of the first spec is `/one/two/three#B`, the identifier of the second spec is `/one/two/three#three`.

As a particular instance of the case of multiple units per file, it is possible to have just one unit definition in the file. This is different from just having a unit term in a file. If the file contains a unit definition, then the word symbol at the left of “=” is part of the unit’s identifier, together with “#” and the file path (relative to the directory in `SWPATH`). If instead the file contains a unit term, then the identifier of the unit is the file path (relative to the directory in `SWPATH`), without any “#” and additional word symbol.

Despite the possibility of having one unit definition in a file, in this manual we use the term “multiple-unit file” to denote a file that contains one or more unit definitions. The term “single-unit file” is instead used to denote a file that only contains a unit term.

As a convenience, a unit in a multiple-unit file with the same name as the file (without the directory and extension) may be referred to with a URI for the file as a whole. For example, in the current case, the identifier `/one/two/three` refers to the same spec as `/one/two/three#three`. This feature supports a style of having one primary unit in a file with auxiliary units that are used to define the primary unit.

## 4.3 Unit Definitions Are Managed Outside of Specware

The `.sw` files are created, deleted, moved, and renamed by directly interacting with the file system of the underlying operating system.

The content of the `.sw` files can be edited with any desired text editor. A possibility is to use the XEmacs in which the Specware Shell is running when Specware is fired up using `Specware4 XEmacs`. The XEmacs-Specware combo can be thought of as a limited Integrated Development Environment (IDE).

Note that unit definitions can be managed without running Specware at all. As described in the next chapter, Specware is used to process unit definitions.



---

# THE SPECWARE SHELL

## 5.1 Overview

Unit definitions are processed by Specware. The user instructs Specware to process units by supplying certain commands. Specware has access, via the Lisp runtime environment, to the underlying file system, so it can access the `.sw` files that define units. The environment variable `SWPATH` determines which `.sw` files are accessed by Specware to find unit definitions.

Processing a unit causes the recursive processing of the units referenced in that unit's term. For instance, if a spec `A` extends a spec `B` which in turn extends a spec `C`, then when `A` is processed, `B` and `C` are also processed. There must be no circularities in the chain of unit dependencies.

Processing causes progress and/or error messages to be displayed in the window containing the Specware Shell. Progress messages inform the user that units were processed without error. Error messages provide information on the cause of errors, so that the user can fix them by editing the unit definitions. If the Specware Shell is running under XEmacs, then, when an error occurs in the definition of some unit, Specware displays the `.sw` file containing the unit term in a separate XEmacs buffer, with the cursor positioned at the point of the erroneous text.

The processing of certain kinds of units also results in the creation of new files as an additional side effect. For instance, Lisp, C or Java programs are constructed by units containing the `generate` operation of Metaslang. A side effect of processing such a unit is that the resulting code is written into a file.

When Specware processes a unit, it saves the processing results into an internal cache, associating the results with the unit's identifier. By using this cache, Specware avoids unnecessary re-computations: it only re-processes the units whose files have changed since the last time they were processed. From the point of view of the final result, this caching mechanism is transparent to the user. However, it improves the performance and response time of the system.

However, under certain circumstances this may lead to the wrong result. Files only need to be processed if they may have changed since the last time they were processed. To determine whether this is the case, the caching mechanism uses the "last modified" date and time of the files. Say there are two files named `mickey.sw` and `minnie.sw`. If the user first lets Specware process `mickey.sw`, and then deletes that file and renames `minnie.sw` to `mickey.sw`, the system will be fooled into assuming that `mickey.sw` does not need to be re-processed. After all, its modification time is that of the original `minnie.sw` file, and so it is older than the last time `mickey.sw` was processed. A likely scenario under which this may happen is when a user copies a file to a back up, modifies the file, has Specware process it, and then restores it by moving the back-up version in its place. All other scenarios that may lead to the wrong results are variations on this theme, replacing a file with one with the same name but different content during a Specware session without adjusting its modification time, or by antedating its modification time.

Clearly, to retain cache integrity, the user is well-advised not to rename, move or delete `.sw` files while a Specware session is in progress. If there is any reason to suspect that the integrity of the cache has become compromised, the Specware Shell command **cinit** will clear the unit cache and thereby restore integrity.

## 5.2 Typechecking Specs

The user can construct specs by explicitly listing the types, ops, and axioms comprising the spec, possibly after importing one or more other specs. When a spec is processed, Specware typechecks all the expressions that appear in the spec. Typechecking means checking that the expressions are type-correct, according to the rules of the Metaslang language.

In general, only some of the ops and variables that appear in an expression have explicit type information. Typechecking also involves reconstructing the types of those ops and variables that lack explicit type information.

Typechecking is an integrated process that checks the type correctness of expressions while reconstructing the missing type information. This is done by deriving and solving type constraints from the expression. For instance, if it is known that an op  $f$  has type  $A \rightarrow B$  then the type of the variable  $x$  in the expression  $f(x)$  must be  $A$ , and the type of the whole expression must be  $B$ .

If the missing type information cannot be uniquely reconstructed and/or if certain constraints are violated, Specware signals an error, indicating the textual position of the problematic expression.

Since the Metaslang type system features subtypes defined by arbitrary predicates, it is in general undecidable whether an expression involving subtypes is type-correct or not. When Specware processes a spec, it assumes that the type constraints arising from subtypes are satisfied, thus making typechecking decidable.

The proof obligations associated with a spec, which are extracted via the Metaslang **obligations** operation, include conjectures derived from the type constraints arising from subtypes. If all of these conjectures can be proved (using a theorem prover) then all the expressions in the spec are type-correct.

## 5.3 Resolution of Unit Identifiers

Unit terms may reference units in the form of unit identifiers. A unit identifier is resolved to the unit's term, which is contained in a `.sw` file. Unit identifiers are either `SWPATH`-based or relative; these two kinds are syntactically distinguished from each other and are resolved in slightly different ways.

### 5.3.1 `SWPATH`-Based Unit Identifier

A `SWPATH`-based unit identifier starts with `/`, followed by a `/`-separated sequence of one or more path elements, where the last separator may be `#`. Examples are `/a/b/c`, `/d`, and `/e#f`.

Specware resolves a `SWPATH`-based unit identifier in the following steps:

1. If the unit identifier contains `#`, the `#` itself and the path element following it are removed, obtaining a `/`-separated sequence of one or more path elements, preceded by `/`. Otherwise, no removal takes place. Either way, the result of this first step is a `/`-separated sequence of path elements preceded by `/`.
2. The `/` signs of the `/`-separated sequence of path elements preceded by `/`, resulting from the previous step, are turned into `""`; in addition, `.sw` is appended at the end. The result of this second step is a (partial) path in the file system.
3. The path resulting from the previous step is appended after the first directory of `SWPATH`. If the resulting absolute path denotes an existing file, that is the result of this third step. Otherwise, the same attempt is made with the second directory of `SWPATH` (if any). Attempts continue until a directory is found in `SWPATH` such that the absolute path obtained by concatenating the directory with the result of the previous step denotes an existing file; such a file is the result of this step. If no such directory is found, the unit identifier cannot be resolved and an error is signaled by Specware.
4. There are two alternative steps here, depending on whether or not the original unit identifier contains `#`.



- (a) This is the case that the original unit identifier does not contain #. If the file resulting from the previous step is a single-unit file, i.e., it contains a unit term, that the final result of resolution. Otherwise, an error is signaled by Specware.
- (b) This is the case that the original unit identifier contains #. The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit identifier cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose path elements (to the left of “=”) is the same as the path element that follows # in the original unit identifier. If no such unit definition is found, the unit identifier cannot be resolved and an error is signaled by Specware. If such a unit definition is found, its unit term (at the right of “=”) is the final result of resolution.

For example, consider a unit identifier `/a/b/c`. Since it does not contain #, the first step does not do anything. The result of the second step is `\a\b\c.sw`. Suppose that `SWPATH` is `C:\users\me\specware;C:\tmp`, that `C:\users\me\specware` does not contain any a subdirectory, and that `C:\tmp\a\b\c.sw` exists. The result of the third step is the file `C:\tmp\a\b\c.sw`. If such a file is a single-unit file, its content is the result of the fourth step.

As another example, consider a unit identifier `/e#f`. The result of the first step is `/e`. The result of the second step is `\e.sw`. Assuming that `SWPATH` is as before and that `C:\users\me\specware` contains a file `e.sw`, the file `C:\users\me\specware\e.sw` is the result of the third step. The file must be a multiple-unit file. Assuming that this is the case and that it contains a unit definition with path element `f`, its unit term is the result of the fourth step.

The directories in `SWPATH` are searched in order during the third step of resolution. So, in the last example, if the directory `C:\tmp` also contains a file `e.sw`, such a file is ignored. This features can be used, for example, to shadow selected library units that populate certain file system directories in `SWPATH`.

For example, suppose that the first directory in `SWPATH` is `C:\specware\libs` and that the directory `C:\specware\libs\data-structures` contains files `Sets.sw`, `Bags.sw`, `Lists.sw`, etc. defining specs of sets, bags, lists, etc. The unit identifier `/data-structures/Sets` resolves to the content of the file `C:\specware\libs\data-structures\Sets.sw`. If the user wanted to experiment with a slightly different version of the spec for sets, it is sufficient to prepend another directory to `SWPATH`, e.g. `C:\shadow-lib`, and to create that slightly different version of the spec for sets in `C:\shadow-lib\data-structures\Sets.sw`. The same unit identifier `/data-structures/Sets` will now resolve to the new version.

### 5.3.2 Relative Unit Identifiers

A relative unit identifier is a `/`-separated sequence of one or more path elements, where the last separator can be #. Examples are `a/b/c`, `d`, and `e#f`. So, `SWPATH`-based and relative unit identifiers can be distinguished by the presence or absence of `/` at the beginning.

The resolution of relative unit identifiers does not depend on `SWPATH`, but on the location of the file where the unit identifier occurs. There are two cases to consider: the unit identifier occurring in a single-unit file and the unit identifier occurring in a multiple-unit file.

Suppose that the relative unit identifier occurs in a single-unit file. Then Specware attempts to resolve the unit identifier in the following steps:

1. If the unit identifier contains #, the # itself and the path element following it are removed, obtaining a `/`-separated sequence of one or more path elements. Otherwise, no removal takes place. Either way, the result of this first step is a `/`-separated sequence of path elements.
2. The `/` signs of the `/`-separated sequence of path elements, resulting from the previous step, are turned into “.”; in addition, `.sw` is appended at the end. The result of this second step is a (partial) path in the file system.
3. “%” signs are left unchanged and `.sw` is appended at the “%”end. Either way, the

4. The path resulting from the previous step is appended after the absolute path of the directory of the file containing the relative unit identifier. If the resulting absolute path denotes an existing file, that is the result of this third step. Otherwise, the unit identifier cannot be resolved and an error is signaled by Specware.
5. There are two alternative steps here, depending on whether the original unit identifier contains # or not.
  - (a) This is the case where the original unit identifier does not contain #. If the file resulting from the previous step is a single-unit file, i.e., it contains a unit term, that is the final result of resolution. Otherwise, an error is signaled by Specware.
  - (b) This is the case that the original unit identifier contains #. The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit identifier cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose path element (at the left of “=”) is the same as the path element that follows # in the original unit identifier. If no such unit definition is found, the unit identifier cannot be resolved and an error is signaled by Specware. If instead such a unit definition is found, its unit term (to the right of “=”) is the final result of resolution.

So, resolution of a relative unit identifier occurring in a single- unit file is like resolution of a SWPATH-based unit identifier, except that the directory where the identifier occurs is used instead of SWPATH.

Suppose, instead, that the relative unit identifier occurs in a multiple-unit file. Then Specware attempts to resolve the unit identifier in the following steps:

1. If the relative unit identifier is a single path element, Specware attempts to find a unit definition with that path element inside the file where the unit identifier occurs. If such a unit definition is found, its unit term is the final result of resolution. Otherwise, the following steps are carried out:
2. If the unit identifier contains #, the # itself and the path element following it are removed, obtaining a “/”-separated sequence of one or more path elements. Otherwise, no removal takes place. Either way, the result of this first step is a \-separated sequence of path elements.
3. The “/” signs of the “/”-separated sequence of path elements, resulting from the previous step, are turned into \; in addition, .sw is appended at the end. The result of this second step is a (partial) path in the file system.
1. The path resulting from the previous step is appended after the absolute path of the directory of the file containing the relative unit identifier. If the resulting absolute path denotes an existing file, that is the result of this third step. Otherwise, the unit identifier cannot be resolved and an error is signaled by Specware.
2. There are two alternative steps here, depending on whether the original unit identifier contains # or not.
  - (a) This is the case that the original unit identifier does not contain #. If the file resulting from the previous step is a single-unit file, i.e., it contains a unit term, that is the final result of resolution. Otherwise, an error is signaled by Specware.
  - (b) This is the case that the original unit identifier contains #. The file resulting from the previous step must be a multiple-unit file, i.e., it must contain a sequence of one or more unit definitions. If this is not the case, the unit identifier cannot be resolved and an error is signaled by Specware. If that is the case, a unit definition is searched in the file, whose path element (at the left of “=”) is the same as the path element that follows # in the original unit identifier. If no such unit definition is found, the unit identifier cannot be resolved and an error is signaled by Specware. If instead such a unit definition is found, its unit term (to the right of “=”) is the final result of resolution.

So, resolution of a relative unit identifier occurring in a multiple- unit file is like resolution of a relative unit identifier occurring in a single-unit file, preceded by an attempt to locate the unit in the file where the identifier occurs, only in case such a unit identifier is a path element.

## 5.4 Command Format

Each Specware Shell command consists of a keyword, the command name, followed by zero or more arguments. Several Specware Shell commands have “optional arguments”: they allow a variable number of arguments; e.g., zero or one. For many such commands, the zero-argument version means: use the last argument of the same kind last used for a Specware Shell command. In other cases, it means: use a default value for the omitted argument. Which commands use which convention is detailed below. Optional arguments are given between square brackets [ and ].

Unit identifiers occurring in a unit term used as a command argument are resolved as described in *Resolution of Unit Identifiers*, where relative unit identifiers are resolved as they would be if the unit term was a single-unit file in the current directory.

A command entered by the user should be typed all on one line. The Return/Enter at the end of the line signals to the Specware Shell that the command must be executed. If the Specware Shell is running under XEmacs, the TAB key can be used for filename completion.

## 5.5 Miscellaneous Commands

The commands described in this section do not process units, but some may influence the way later processing commands work.

A terse description of all Shell commands is produced by the help command:

```
help
```

With an argument:

```
help <command-name>
```

it shows a description of just that command.

The pathname of the current directory is shown on the putput by the following command:

```
cd
```

The change-directory command:

```
cd <directory>
```

sets the current directory (folder) to the argument, which must be a valid pathname for a directory in the file system, either absolute or relative to the present directory. To move one level up, to the parent of the current directory, the special notation “.” can be used. The full pathname of the new current directory is then displayed. This influences the subsequent resolution of unit identifiers if “.” is on the SWPATH. With no argument, the command **cd** just shows the current directory.

Two commands allow listing Specware files:

```
dir
```

list the .sw files in the current directory, while

```
dirr
```

(for DIR Recursive) also lists those in sub-directories.

The value of the SWPATH environment variable is shown via the following command:

```
path
```

The value of the `SWPATH` environment variable is changed via the following command:

```
path <dir>;<dir>;...;<dir>
```

The argument must be a semicolon-separated list of absolute directory paths of the underlying operating system. For example, in order to set `SWPATH` to `C:\users\me` it is necessary to write **path C:\users\me**.

Changes to `SWPATH` only apply to the currently running Specware session. If Specware is quit and then restarted, `SWPATH` loses the value assigned to it during the previous session, reverting to its default value.

## 5.6 Processing a Unit

The command to process a unit is:

```
proc [ <unit-term> ]
```

The argument can be any unit term: a simple unit identifier, a diagram colimit, a proof term, and so on.

If the argument is not a syntactically valid unit term, or some unit identifier in the term fails to resolve as explained in *SWPATH-Based Unit Identifier*, an error is signaled by Specware.

Otherwise Specware parses and elaborates the unit term that results from resolution. Parsing and elaboration carry out the computations to construct the unit; they are Specware’s “core” functionality. Parsing and elaboration implement the semantics of the Metaslang language.

In this process, Specware performs further checks on the requirements as stated in the language manual, such as non-ambiguity of names. If any errors are found, they are signaled

If the unit term references other units, Specware recursively resolves the unit identifiers and parses and elaborates their unit terms.

Finally, if all went well, Specware typechecks the unit resulting from the elaboration process, if applicable, and signals any errors detected.

The elaboration of some unit terms may have side effects: code generation; prover invocation. This is only done if no error was encountered. Code can also be generated directly from the Specware Shell using the **gen-Language** commands. For proving properties in specs, see ‘**Proving Properties in Specs**’.

Without argument, the **proc** command re-processes the last unit term given. It is an error if no unit term was given before.

It is also possible to process a multiple-unit file all at once:

```
proc <multi-unit-identifier>
```

The unit identifier must not contain “#” and the file must not contain a unit of the same name as the file (without the directory and extension). Specware attempts to resolve the unit identifier. If it is a relative unit identifier, it is resolved as if it occurred inside a single-unit file in the current directory. However, the file obtained at the third step must be a multi-unit file, and not a single-unit file. If it is indeed a multi-unit file, Specware parses and elaborates all the unit definitions inside the file.

The command **proc** may be abbreviated to **p**.

If the `<unit-term>` is a proof term, i.e. it begins with **prove**, then the **proc** or **p** may be omitted, so:

```
prove <claim-name> in <unit-term>
```

is the same as

```
proc prove <claim-name> in <unit-term>
```

The values of processed units are kept in the unit cache. To clear the unit cache, as mentioned before, use:

```
cinit
```

(Cache INITialize).

## 5.7 Showing a Unit

When a unit definition is elaborated, a unit value is produced. For example, a spec is essentially a set of types, ops, and axioms. A spec can be constructed by means of various operations in the Metaslang language, but the final result is always a spec, i.e., a set of types, ops, and axioms.

The command for showing unit values is:

```
show [ <unit-term> ]
```

As for **proc** the argument can be any unit term: a simple unit identifier, a diagram colimit, a proof term, and so on, and a missing argument means: use the last argument supplied in a unit-term position. However, unlike for **proc**, the argument can not be a multi-unit identifier.

The unit term is processed as for the **proc** command. If no error occurred, the unit value resulting from elaborating the unit term is shown on the output. However, an attempt is made to keep imported specs as import declarations, instead of expanded in the output. If the argument was already a spec form, the output may look different in several ways: white space may be different, declarations may have been added or re-ordered, and explicit qualifications may have been added.

If the *[unit-term]* is an obligation term then the show may be omitted. For example,

```
obligations <unit-term>
```

or its abbreviation

```
oblig <unit-term>
```

is the same as

```
show obligations <unit-term>
```

To show imported specs expanded in place, use:

```
showx [ <unit-term> ]
```

(for SHOW eXpanded).

“Showing” a proof unit has the same effect as just processing it; the elaboration of a proof unit is only in the side effects.

## 5.8 Generating Proof Units

Two Specware Shell commands facilitate the creation of proof units. The first is:

```
punits [ <unit-identifier> [ <filename> ] ]
```

in which the unit identifier must be that of a single spec term. Executing the command then generates proof units for all the conjectures, theorems and proof obligations of the spec resulting from elaborating that spec term.

These proof units are written to a file that can then be processed to attempt proving all the conjectures in the spec. The proof file can also be edited to add `usings` and `options` to the proof units. By default, the proof units are written in a file obtained from the unit identifier given to the **punits** command. For example, if the unit identifier is `/dir1/dir2/foo`, then the proof units are written to `/dir1/dir2/foo_Proofs.sw`. Optionally, the file for the proof units to be written to can be given as the second argument to the **punits** command.

Using **punits**, proof units are generated not just for the conjectures explicitly present in the spec, but also for all non-local conjectures for the spec. The user can use the command:

```
lpunits [ <unit-identifier> [ <filename> ] ]
```

to generate a proof-unit file with only the “local” conjectures.

## 5.9 Evaluating Expressions

“Constructive” expressions – i.e., using only constructively defined types and ops – can be evaluated directly from the Specware Shell. Evaluating an expression requires a “context” to be set, which is a spec containing the definitions of the relevant types and ops. For example, in the context of:

```
spec
  def f x = 2*x+1
  def u = 6172
endspec
```

The evaluation context can be set by the shell command:

```
ccontext [ <spec-term> ]
```

As usual, if the argument is absent, it indicates the last term processed, which must elaborate to a spec.

Once a context has been set, a Metaslang expression can be evaluated by:

```
eval [ <expression> ]
```

which results in the value of the expression being shown, insofar as possible: some types of values, in particular functions, have no “printable” representation. Apart from that, values are shown using Metaslang syntax; for example, **eval** `[100, 2*100]` shows `[100, 200]`. The evaluation is done by a built-in Metaslang interpreter.

The command **eval** may be abbreviated to **e**.

Instead of using the built-in Metaslang interpreter, it is also possible to evaluate Metaslang expressions from the Specware Shell command line as translated into Lisp. Unless no user-defined types and ops are used (as in the expression `2+2`), this requires that Lisp code has already been generated for the context spec (see *Generating Lisp Code*) and that the Lisp file has been loaded (see *Auxiliary Commands for Lisp*). The command is:

```
eval-lisp [ <expression> ]
```

which also results in the value of the expression being shown, but now using Lisp syntax; for example, **eval-lisp** `[100, 2*100]` shows `(100 200)`. For expressions whose evaluation is very computation-intensive, this method of evaluation can be substantially faster than using the interpreter.

## 5.10 Generating Lisp Code

Lisp code can be generated by constructing a Metaslang unit containing a target-code term (using **generate lisp**) and by processing such unit via the **proc** command.

The Specware Shell provides a command to accomplish the same result without actually creating a separate Metaslang unit. The command is:

```
gen-lisp [<spec-term> [<filename>] ]
```

The spec term is processed by Specware. If this argument is missing, the last spec term processed is used. If the spec is successfully processed, Specware generates Lisp code from it (according to the semantics of **generate lisp**) and deposits the resulting code into the file whose path is given by the filename. The `.lisp` file extension can be omitted.

The filename to `gen-lisp` is also optional. If it is not given, a file name is inferred. If the spec term given as argument is a unit identifier, Specware deposits the generated code into the file `U.lisp`, where `U` is the rightmost path element comprising the unit identifier. The `U.lisp` file is deposited in a `lisp` subdirectory immediately under the directory of the file containing the unit term of the spec identified by the unit identifier given as argument to `gen-lisp`.

For example, suppose that the first directory in `SWPATH` is `C:\users\me\specware` and that a spec is defined in the single-unit file `C:\users\me\specware\two\A.sw`. If the user gives the command:

```
gen-lisp /two/A
```

the Lisp code is deposited into the file `C:\users\me\specware\two\lisp\A.lisp`.

As another example, suppose that `SWPATH` is as before and that a spec is defined in the multiple-unit file `C:\users\me\specware\two\F.sw`, and that `B` is the path element associated with the spec. If the user then gives the command:

```
gen-lisp /two/F#B
```

the Lisp code is deposited into the file `C:\users\me\specware\two\lisp\B.lisp`.

If the spec term given as argument is not a unit identifier, Specware deposits the generated code in a file under `C:\tmp\sw\lisp`, if possible. In all cases the name of the Lisp file is shown on the output.

The shell command:

```
lgen-lisp [ <spec-term> [ <filename> ] ]
```

is like **gen-lisp**, but generates code only for the local definitions of the spec and not any of the imports. It is intended for incremental development. Note that if you have not generated code for the imported specs and loaded it, trying to run the code generated by this command will give undefined function errors. Also, if the spec is unqualified but it is imported into a spec that is qualified, the package used will be `:SW-USER` instead of the package of the qualifier. To avoid this problem, qualification can be added to the spec.

### 5.10.1 Generating Java Code

As an experimental feature, Specware provides the possibility to generate Java code from constructive Metaslang specs. The Java code generator can either be called from the Specware Shell or using the `generate` construct inside a `.sw` file. In both cases, an additional “option” spec can be supplied, which is used to specify certain parameters that govern aspects of code generation. For the format of the option spec, see [The Option Spec](#).

The command has the form:

```
gen-java <spec-term> [<option-spec-term>]
```

where the result of elaborating the spec term gives the spec to be translated into Java and the second spec term gives the option spec (see below).

## The Option Spec

The option spec is used as an attribute store to be able to control certain parameters used by the Java code generator. The option spec is a regular Metaslang spec. The parameters are given by constant ops defined inside the option spec. The following list contains the op names and types that are currently interpreted as parameters by the Java code generator:

Op name & type	Used as	Default value
package : String	Name of the Java package for all generated Java class files. The package name also determines the relative path of the generated . java files (see the basedir parameter)	"specware.generated"
basedir : String	The base directory used for the generated Java class files. The full path is determine by this parameter and the relative path derived from the package name. For instance, if the value of basedir is the string "/a/b" and the package name is c.d.e, then the generated Java class files would go into the directory /a/b/c/d/e.	". "
public : List String	The list of op names that are to be declared as public in the generated Java code. Only unqualified identifiers can be used in this list. The ops in this list determine the “entry points” into the generated Java code, if it is embedded in another Java application.	[]
"imports : List String"	The list of imports that are to be used for all generated Java class files. Each element of this list has the usual format of the argument used by Java’s import statement; e.g., "java.util.*"	[]

Example option spec:

```
spec
  def package = "test.sw.gen"
  def imports = ["java.util.*"]
  def public = ["test2"]
  def basedir = "~/myjavaapps"
endspec
```

If no option spec is specified in the `gen-java` command, default values are used for all option parameters.

## Translation of Inbuilt Ops

The following table shows the translation of some inbuilt Metaslang ops into Java:

Metaslang	Java
String.writeLine(t) String.toScreen(t)	System.out.println(t)
String.concat(t1,t2) t1 ++ t2	t1 + t2
String.newline	System.getProperty("line.separator")
String.length	t.length()
String.substring(s,n1,n2)	s.substring(n1,n2)
Nat.toString(n) Nat.natToString(n) Nat.show(n) Integer.toString(n) Integer.intToString(n) Integer.show(n)	String.valueOf(n)
Nat.stringToNat(s) Integer.stringToInt(s)	Integer.parseInt(s)
t1 && t2	t1 && t2
t1    t2	t1    t2
t1 => t2	t1 ? t2 : true
t1 <=> t2	t1 ? t2 : !t2



## Metaslang/Java Interface

In order to use Java methods and classes inside a Metaslang spec, the following conventions are used by the Java code generator:

**Java Classes** In order to use Java classes as types inside Metaslang, you have to declare the type without a definition and add corresponding Java import statements using the option spec (see above). Example: use of the Java class `java.util.Vector` In the spec for which code is generated:

```
...
type Vector
...
op myvectorop: Vector -> Nat
def myvectorop(v) = ...
...
```

In the option spec:

```
...
def imports = [ ..., "java.util.*", ... ]
...
```

The code generator interprets all types without a definition as base types, so that in this case the `op myvectorop` becomes a static method in the generated `Primitive` class.

**Accessing External Java Instance Methods** Instance methods as well as static class methods can be accessed from inside Metaslang specs using the following convention: Assume, we want to use some instance method `epi(args)` defined in Java class `Tecton`. First, the class must be made known to Metaslang by providing a type declaration for the class. Then, an `op epi` must be declared with a signature that corresponds to the method's signature, but with an additional parameter preceeding the others. The type of that parameter must be the class type:

```
type Tecton
op epi: Tecton * T1 * ... * Tn -> T
```

where `T1 * ... * Tn -> T` is the original signature of `epi` without the additional parameter. The `Ti`'s are the translated Metaslang types that correspond to the Java types occurring in `epi`'s signature; see the table below for the type correspondence. In the Metaslang code, a call to the instance method is created by the Java code generator whenever `epi` is applied:

```
def mycode(...) =
  ...
  let b : Tecton = ... in
  ...
  ... epi(b, arg1, ... argn) ...
```

Note, that a definition term must not be given for `epi`. Limitation: using `epi` as a function symbol in higher-order contexts will not yield the expected result.

**Accessing External Java Class Methods** Accessing Java class methods is very similar to instance methods, with the difference that instead of the type of the first argument, the qualifier of the `op` declaration is used to determine the class name. Therefore, in general, it is not necessary to declare the class as a type. Assume we want to access to class method `Math.abs()` from the Java library. We therefore declare the `abs` operator in Metaslang as follows:

```
op Math.abs: Integer -> Nat
```

The code generator will then generate a call to the static method `Math.abs()` whenever it is used in the Metaslang spec. The access to static methods has lower priority than the access to instance methods: if the first argument is a user type that is not defined in the spec, than the instance call is generated. In other words, a static

method in *class* A with a first argument of *type* A will not be accessible from Metaslang. The latter situation is not very common, and in practice this does not constitute a limitation of the Metaslang-Java interface.

**Accessing Java Constructors** Accessing Java constructors follows the same principle as for class methods. The difference is that on the Metaslang side, an op with a name having the prefix `new` and an appropriate result type must be declared. For instance, the Java class `Vector` declares a constructor with no arguments. If we want to use that in Metaslang, we have to provide the following declarations:

```
type Vector
op Vector.new: () -> Vector
```

Whenever `Vector.new()` is used as a term in the Metaslang spec, a call to the corresponding Java constructor in the `Vector` class is generated. If the class has multiple constructors with different parameter lists, multiple `new` ops can be declared in the Metaslang spec with different suffixes (e.g., `new_2`). The Java code ignores the suffixes, but they are essential for Metaslang, which does not allow the redefinition of ops with different signatures. In general, if multiple methods and constructors from a class in the Java library need to be accessed in a Metaslang spec, it is a good idea to structure them using the `qualifying` feature of Metaslang. For instance:

```
Vector qualifying spec
  type Vector
  op new: () -> Vector
  op add: [a] Vector * a -> Vector
  op size: Vector -> Nat
endspec
Math qualifying spec
  op max: Integer * Integer -> Integer
  op min: Integer * Integer -> Integer
  ....
endspec
```

and then importing the specs into the application spec that uses it. Future versions of the Specware system will provide a utility to convert a given Java class into a spec following the above conventions.

## Type Conversion between Java and Metaslang

The following table shows the conversion of Java types to Metaslang, which can be used when accessing Java methods from Metaslang

Java	Metaslang
int	Integer
boolean	Bool
char	Char
void	()
byte short float double	not implemented
Any Java class name	Metaslang type with the same name (type must be declared in the spec)

### 5.10.2 Generating C Code

As an experimental feature, Specware provides the possibility to generate C code from constructive Metaslang specs. The C generator has been tested to work under Linux as well as Windows, the latter using the Cygwin DLL (see [www.cygwin.com](http://www.cygwin.com)). The C code generator can either be called from the Specware Shell or using the Metaslang `generate` construct inside a `.sw` file. In both cases, an additional parameter can be supplied specifying the basename of the C source and header files constituting the generated C code.

The shell command is:

```
gen-c <spec-term> [ <c-file-basename> ]
```

where the result of elaborating the spec term gives the spec to be translated into C and the second optional argument is the basename.

For example:

```
gen-c SortImpl Quicksort
```

takes the spec in file `SortImpl.sw` and translates it into the C files `Quicksort.h` and `Quicksort.c`

## Generating, Compiling and Running the generated C code using “make”

The easiest and recommended way of generating C code and compiling it is by using the Specware Shell command

```
make <spec-term>
```

This requires that the GNU utilities `make` and `gcc` (or equivalent utilities) are installed. The `make` utility must reside in a directory on the Windows system path, or be referenced by the environment variable `SPECWARE4_MAKE` (see below). The `gcc` utility must likewise reside in a directory on the Windows system path, or alternatively the definitions of make variables `CC` and `LD` in the `Makerules` file (see below) must be made to reference the name or pathname of an installed C compiler.

The Specware Shell `make` command does the following things:

- it invokes the `gen-c` command on the given spec term and uses the name of the unit-id as file name for the generated C code (`#`'s are replaced by `_`'s). For example:

```
make Layout#Multi
```

would invoke `gen-c Layout#Multi Layout_Multi.c`

- if the C code generation has been successful, a customized Makefile is generated into `swcmake.mk`. This file will include references to the built-in `Makerules` and define the targets and dependencies in a way that it compiles and executable with the same name as the generated C files with their suffixes removed; e.g., for the above example the name of the executable would be `Layout_Multi`. By convention, if a file named `B_main.c` or `B_test.c` exists, where `B` is a the basename for the generated C files, it will be automatically included in the build process; `B_test.c` is only used if `B_main.c` does not exist. For the above example this would mean that, if a file named `Layout_Multi_main.c` exists, it will be included in the build. In addition to the built-in `Makerules` file, the generated Makefile `swcmake.mk` will also include a unit-specific Makefile in the current directory called `B.mk` if such a file exists; e.g., in the above example, `Layout_Multi.mk`. This file can be used to set the make variables `CFLAGS` and `USERFILES`, which are used as follows:

one	two
<code>CFLAGS</code>	the value of the <code>CFLAGS</code> variable is used in calls to the C compiler ( <code>gcc</code> ) and usually contains example-specific flags, e.g., optimizer flags. Example: <code>CFLAGS = -O3</code>
<code>USERFILES</code>	the value of the <code>USERFILES</code> make-variable is used in calls for the final compilation and linking of the executable. It usually lists additional C files ( <code>.o</code> and/or <code>.c</code> files) that the example needs to be a fully stand-alone application.

Other make variables that are used in the generated/predefined rules are `LD_FLAGS` (which can be used to add additional libraries, etc.), `CPPFLAGS` (see below), and `USEGC` (see below).

- Finally, the Unix “make” command is called with the generated Makefile `swcmake.mk` as top-level Makefile. By default, the command called is “make”, which requires a program with this name to be available in the current system path setting. The system environment variable `SPECWARE4_MAKE` can be used to overwrite this default behavior: if `SPECWARE4_MAKE` is set, its value is used as the command to be called.

## Compiling and Running the generated C code without using “make”

The generated C code is designed to contain as few references outside the generated code as possible, but there are still some built-in routines that are referred to. For that reason, the C compiler needs a few extra arguments specifying system include paths and the location of the garbage collector library that is used in the generated code. The easiest way of compiling the generated code is by using a Makefile and including the supplied C generator system Makefile in it. The corresponding include statement in a user Makefile would then be as follows:

```
include $(SPECWARE4)/Library/Clib/Makerules
```

where environment variable `SPECWARE4` is set to the installation directory of Specware, e.g., `C:/progra~1/Specware4.2`. This `Makerules` file sets the `CPPFLAGS` and `CFLAGS` make variables to include the paths and library necessary for successfully compiling the generated code. If additions to these variables need to be made, one can either define the augmented variable before the above include statement in the Makefile, or use the `:=` assignment after the include statement to prevent “make” from recursively processing the variable. For example:

```
CPPFLAGS := -g -pg $(CPPFLAGS)
```

is a valid statement for augmenting the `CPPFLAGS` variable after the include statement. See:

```
%SPECWARE4%\Languages\MetaSlang\CodeGen\C\Examples\Makefile
```

for an annotated example Makefile.

## Garbage Collector

By default, the generated C code generates calls to the public-domain Boehm garbage collector (see [www.hpl.hp.com/personal/Hans\\_Boehm/gc/](http://www.hpl.hp.com/personal/Hans_Boehm/gc/)). The library needs to be built once on a fresh Specware4 tree and will then be used by the Specware-generated C code. The easiest way to build the gc-library is described in the example Makefile mentioned above: simply add the variable `$(GCLIB)` to the list of dependencies in the main Makefile target. Alternatively, this can be done by hand by changing to the directory `%SPECWARE4%\Languages\MetaSlang\CodeGen\C\Clib\gc6.6` and then running “make”. After successful completion of this command, a file named `gc.a` should be present in that directory.

To disable the garbage collector, simply put the variable definition:

```
USEGC = no
```

in front of the line including the above `Makerules` file. This will prevent the generated code from calling the allocation function of the garbage collector and the garbage collector library will not be bound to the executable.

## Supplying a C “main” function

To create a stand-alone C application using the Specware-generated code, the user has to supply a main function. This can be done either by directly defining an unqualified Metaslang operator `main` like this:

```
op main: () -> ()
def main () ...
```

or by hand-coding a C function `main()` in a separate C file, from where the Specware-generated code is called. Passing command-line arguments is not yet supported when defining a Metaslang `main` operator directly. See the Examples directory for examples of both a hand-written “main” C function that calls the generated code, and a Metaslang definition of `op main`.

### 5.10.3 Generating Haskell Code

As an experimental feature, Specware provides the capability of generating Haskell code from constructive Metaslang specs. Currently all of the basic language is supported except for unnamed records.

The Haskell code generator can either be called from the Specware Shell or using the emacs interface.

There are two shell commands for generating Haskell, one that creates modules that export everything and one that creates modules that only export what is needed by the given spec, with unneeded ops omitted. The latter is useful when applied to the top-level spec of interest. The former is useful when you want code for lower-level specs that may be used in different contexts.

The first shell command, that exports all imports, is:

```
gen-haskell <spec-term>
```

where the result of elaborating the spec term gives the spec to be translated into Haskell. The **gen-haskell** command can be abbreviated to **gen-h**. Inside the emacs interface it is simpler to use the `C-c h` command (`sw:convert-spec-to-haskell`) which has the same effect but also visits the generated Haskell file. For a single file spec, the generated Haskell file has the same name as the Specware file with extension “hs” instead of “sw” and is put in the “Haskell” subdirectory of the directory containing the spec. A spec in a multi-spec file also has the local name of the spec appended to the file name. Haskell code is generated for any imported specs if necessary. The Haskell code for the Base library is pre-generated and stored in the directory `%SPECWARE4%\Library\Haskell`, so this directory must be on your Haskell system’s load path in order to process the generated Haskell files. The Base library and its translation into Haskell provides good examples of the Haskell generator and how to control it using the pragmas documented below.

The second shell command for use with top-level specs is:

```
gen-haskell-top <spec-term>
```

where the result of elaborating the spec term gives the spec to be translated into Haskell. The **gen-haskell** command can be abbreviated to **gen-ht**. Inside the emacs interface it is simpler to use the `C-c H` command (`sw:convert-top-spec-to-haskell`) which has the same effect but also visits the generated Haskell file. This also generates modules for all imported spec.

The names of the generated Haskell modules are the same as that for the **gen-haskell** command, but the files generated for specs in different directories are put in sub-directories of the top-level Haskell directory. For specs in sub-directories of the top-level spec, the generated module files are in the corresponding sub-directory of the top-level Haskell directory. For example, given the command `gen-haskell-top /A/B/C/Main.sw`, the top-level module is stored in file `/A/B/C/Haskell/Main.hs` and if this spec imports (possibly indirectly) the spec `/A/B/C/D/E/F.sw` the corresponding module is generated as `/A/B/C/Haskell/D/E/F.hs`.

For specs in parent or sibling directories of the top-level spec, a sub-directory called “Par\_\_i” is created in the top-level Haskell directory representing the common parent directory where *i* is the number of directory levels that the parent directory is above that containing the top-level spec, and sub-directories created below that corresponding to the path of the sibling spec. In the example for **gen-haskell-top /A/B/C/Main.sw**, if this spec imports spec `/A/W/X/Y.sw` the corresponding module is generated as `/A/B/C/Haskell/Par__2/W/X/Y.hs`.

#### Translation of Specware Names

Specware unqualified type and op names are translated as is, excepted that the first letter is down- or upper-cased respectively as necessary to comply with Haskell’s lexical rules. Qualified names have the qualifier appended at the beginning of the name, separated from the primary name by two underbar characters, changing the first-letter case if necessary. For example, the type `A.B` is translated to `A__B` and the op `A.b` is translated to `a__b`. However, the user may provide a pragma to override this default translation of an op. The pragma must occur immediately after the op definition if it is unnamed or anywhere if names. The unnamed version has the form:

```
#translate Haskell -> desiredName #end
```

where `desiredName` is the name you want to appear in the Haskell translation. A named pragma has the actual op (or type) name occur after `#translate Haskell`, so the pragma may occur anywhere in the file. E.g.:

```
#translate Haskell Qual.opName -> desiredName #end
```

## Translation Tables

You can connect a Specware spec with an existing Haskell module by providing a `-morphism` translation table within the spec.

A translation table for Specware types and ops is introduced by a line beginning `#translate Haskell -morphism` followed optionally by one or more Haskell module names (which will be imported into the translated spec), and terminated by the string `#end`. Each line gives the translation of a type or op. For example, for the Specware List spec we have:

```
#translate Haskell -morphism List
  type List.List      -> []
  Nil                 -> []
  Cons                -> :                Right 5
  List.List_P         -> list_all
  List.length         -> length
  List.@              -> !!              Left 9
  List.empty          -> []
  List.empty?         -> null
  List.in?            -> elem             Infix 4
  List.nin?           -> notElem          Infix 4
  List.prefix         -> take             curried  reversed
  List.removePrefix   -> drop            curried  reversed
  List.head           -> head
  List.last           -> last
  List.tail           -> tail
  List.butLast        -> init
  List.++             -> ++              Left 5
  List.|>             -> :                Right 5
  List.update         -> list_update     curried
  List.forall?        -> all
  List.exists?        -> any
  List.filter         -> filter
  List.zip            -> zip              curried
  List.unzip          -> unzip
  List.zip3           -> zip3            curried
  List.unzip3         -> unzip3
  List.map            -> map
  List.isoList        -> map
  List.reverse        -> reverse
  List.repeat         -> replicate       curried  reversed
  List.flatten        -> concat
  List.findLeftMost   -> find
  List.leftmostPositionSuchThat -> findIndex curried  reversed
  List.positionsSuchThat -> findIndices  curried  reversed
  List.positionsOf    -> elemIndices    curried  reverse
#end
```

A type translation begins with the word `type` followed by the fully-qualified Specware name, `->` and the Haskell name. Note that by default, sub-types are represented by their super-type.

An op translation begins with the fully-qualified Specware name, followed by `->` and the Haskell constant name. If the Haskell constant is an infix operator, then it should be followed by `Left` or `Right` depending on whether it is left or right associative and a precedence number. Note that the precedence number is relative to Haskell's precedence ranking, not Specware's. Also, an uncurried Specware op can be mapped to a curried Haskell constant by putting `curried` after the Haskell name, and a binary op can be mapped with the arguments reversed by appending `reversed` to the line.

## Translation To Type Class Instances

A Specware type can be translated to be an instance of a Haskell type class by including within the spec an `-instance` pragma. The name of the typeclass to instantiate comes immediately after the `-instance` specifier, followed by the name of the Specware type. On subsequent lines are translations of Specware ops to the functions of the type class in the same syntax as for translation tables above. For example, a monad defined in Specware can be specified to be translated to an instance of the Haskell monad typeclass using the pragma:

```
#translate Haskell -instance Monad Env
  monadBind -> >>= Left 1
  return -> return
#end
```

Note that if there are any ops to be translated to Haskell functions in a type class where you want to use the Haskell definition rather than the Specware definition, then you should use a `-morphism` pragma instead of an `-instance` pragma. In particular, to translate to an existing Haskell monad, you use a `-morphism` pragma.

## Connecting to Haskell IO Monad

The spec `/Library/Base/BasicIO` defines basic IO and exception operations on a monad and includes a morphism to Haskell's IO monad. The monad type `IO`. `IO` corresponds to Haskell IO monad type.

## Strictness Pragas

Both `coproduct` types and ops may have `-strict` in their associated pragmas to cause the translator to generate strictness annotations in the resulting Haskell.

For example, the type specification:

```
type Pair(a,b) = Pair(a * b)
#translate Haskell -strict #end
```

produces:

```
data Pair a b = Pair__Pair !a !b
```

I.e. the pragma causes the addition of `!` before all the constructor fields.

In the case of an op definition, the `-strict` pragma causes the addition of `$!` for all applications to force evaluation of function arguments before calling the function. For example:

```
op ff(i: Int): Int = f(f i)
#translate Haskell -strict #end
```

produces:

```
ff :: Int -> Int
ff i = f $! f $! i
```

Like other pragmas associated with particular types and ops, the pragma may be named rather than immediately following the type or op.

### Header Pragmas

A `-header` pragma specifies text to go at the beginning of the generated Haskell module. For example, the pragma:

```
#translate Haskell -header
{-# OPTIONS -fno-warn-duplicate-exports #-}
#end
```

will add the text `{-# OPTIONS -fno-warn-duplicate-exports #-}` at the beginning of the generated Haskell file.

## 5.11 Auxiliary Commands for Lisp

When developing a Specware application you may generate Lisp code for your application, and then load and test it in the same image in which the Specware system is running. Then if you make a modification to a single spec you can use the **lgen-lisp** command to just generate the generated code for the modification. This command is also useful when you just want to see the generated code for a particular spec.

To load Lisp files from the Specware Shell, the following commands can be used:

```
ld [ <lisp-filename> ]
```

The generated Lisp files can also be compiled from the Specware Shell, but of course only if the running Lisp system has compiling enabled:

```
cf [ <lisp-filename> ]
```

(for Compile File) will compile a Lisp file, and

```
cl [ <lisp-filename> ]
```

compiles and loads it.

## 5.12 Finally

To terminate a Specware session:

```
exit
```

or, equivalently,

```
quit
```



---

# PROVING PROPERTIES IN SPECS

Specware provides a mechanism for verifying the correctness of properties either specified in Metaslang specs or automatically generated as proof obligations arising from refinements or typechecking. Currently Specware comes packaged with the Snark first-order theorem prover. Interaction with Snark is through the proof unit described below.

## 6.1 The Proof Unit

The user invokes the Snark theorem prover by constructing and processing a proof term. A typical proof term is of the form:

```
prove f_stable in Stability
  with Snark
  using stable_chn, f_defn
  options "(use-paramodulation t)
          (use-resolution nil)
          (use-hyperresolution t)"
```

In this proof term, *Stability* must be a spec-valued unit term, and *f\_stable*, *stable\_chn*, and *f\_defn* must all be names of claims (i.e. axioms, conjectures, or theorems) that appear in the spec resulting from elaborating that unit term. If this proof term is in the single-unit file `pruf.sw`, then issuing the command `proc pruf` will result in first translating *stable\_chn*, *f\_defn* and *f\_stable* to Snark formulas, and then invoking the Snark prover to try to prove *f\_stable* from the hypotheses *stable\_chn* and *f\_defn*, using the options in the `options` list. As of this release claim names are qualified. If the user does not explicitly use a qualifier then all properties that have that name will be used, regardless of their qualification. Note also that Specware does not require property names to be unique. If there are more than one property that has the same name as a claim in the `using` clause then they will all be sent to Snark.

To avoid circular proofs, the claims used as hypotheses – *stable\_chn* and *f\_defn* in the example – are required to appear earlier in that spec than the claim to be proved – *f\_stable* in the example. Most users will omit the `options` part. Additionally, the `using` part can be omitted as well. In that case all the claims that appear in the spec term before the claim to be proved will be used as hypotheses in the proof.

The `with` clause is used to specify which prover to be used. If the `with` clause is omitted which is typically the case, then before invoking Snark, the claim will be sent to a simple integer linear inequality decision procedure based on the Fourier Motzkin method. If this fails then Snark will be invoked. Optionally the user can specify either, *Snark* or *FourierM* to explicitly state which prover he wants used. Note that the inequality decision procedure does not make use the properties specified in the `using` clause.

After Snark completes, Specware will report on the success or failure of the Snark proof.

## 6.2 Implicit Axioms

Metaslang `type` and `op` declarations as well as `op` definitions give rise to implicit axioms that need to be sent to Snark. In the case of `type` and `op` declarations, these axioms correspond to the semantics of the Metaslang type system as described in the language manual. In the case of an `op` definition, these axioms correspond to lifting embedded conditionals from the body of the definition as well as translating Metaslang's pattern matching to first-order logic. Note that a single definition can give rise to multiple axioms. If the `using` clause of the proof unit is omitted then these axioms will automatically be sent to Snark. However, if an explicit `using` clause is used then only those axioms that are explicitly mentioned are sent to Snark. In this case to send the axioms corresponding to `op` `op_name` to Snark the user would include the axiom `op_name_def` in the `using` clause. Similarly to include the axioms for `type` `type_name` the user should include the axioms `type_name_def`.

## 6.3 Proof Errors

Specware will report an error if the claim to be proved does not occur in the spec, or if not all claims following `using` occur in the spec before the claim to be proved.

Snark will likely break into Lisp if the user inputs an incorrect option.

## 6.4 Proof Log Files

In the course of its execution Snark typically outputs a lot of information as well as a proof when it finds one. All this output can be overwhelming to the user, yet invaluable in understanding why the proofs succeeded or failed. To deal with all this output Specware redirects all the Snark output to log files. In our example above, which executed a proof in the file `pruf.sw`, Specware will create a subdirectory called `Snark` at the same level as `pruf.sw`. In that directory a log file called `pruf.log` will be created that contains all the Snark output. In this same directory a specware file called `pruf.sw` will also be generated that includes an expanded version of the original spec containing the theorem to be proved. The original spec needs to be expanded before being passed to Snark because the Snark's logic is different from Specware's, lacking polymorphism and higher-order functions.

## 6.5 Multiple Proofs

As there can be multiple units per file, there can be multiple proof units in single file. For example, in file `pruuf.sw` we could include more than one proof unit, as follows:

```
p1 = prove prop1 using ax1, ax2
p1a = prove prop1 using ax3
p2 = prove prop2
```

In this case `proc pruuf` will invoke Snark three separate times, writing three different log files. In this case an additional subdirectory will be created under `Snark`, called `pruuf`. The three log files will then be: `Snark/pruuf/p1.log`, `Snark/pruuf/p1a.log`, and `Snark/pruuf/p2.log`.

## 6.6 Interrupting Snark

As any first-order prover is wont to do, Snark is likely to either loop forever or run for a longer time than the user can wait. The user can provide a time limit for Snark by using an appropriate option. However, there are likely to be times when the user wants to stop Snark in the middle of execution. The user can do this by typing `Cntrl-C` `Cntrl-C` in

the *common-lisp* buffer. This will then interrupt Snark and place the user in the Lisp debugger. The user can exit the debugger by issuing the `:pop` command. A log file will still be written that can be perused if so desired.

## 6.7 The Prover Base Library

Specware has a base library that is implicitly imported by every spec. Unfortunately, the axioms in this library are not necessarily written to be useful by Snark. Instead of having Snark use these libraries we have created a separate base library for Snark. This library is located at `/Library/ProverBase/`. For each spec in `/Library/Base/` there is a corresponding prover spec that shadows it. This prover base spec imports the `'op` and `'type` declarations from the corresponding base spec, and substitutes for the original definitions and axioms, axioms that are more appropriate for sending to Snark. The axioms in these specs are automatically sent to Snark as part of any proof.

## 6.8 The Experimental Nature of the Prover

Our experience with the current prover interface is very new and as such we are still very much experimenting with it and don't expect it to be perfect at this point in time. Many simple theorems will be provable. Some that the user thinks should be might not, and the user will be required to add further hypothesis and lemmas that may seem unnecessary. We are currently working on making this interface as robust and predictable as possible, and welcome any feedback the user can offer.

One area where the user can directly experiment is with the axioms that make up the `ProverBase`. The axioms that make up an effective prover library are best determined by an experimental evolutionary process. The user is welcome to play with the axioms in the `ProverBase`, by adding new ones or changing or deleting old ones. Keep in mind the goal is to have a single library that is useful for a wide range of proofs. Axioms that are specific to different proofs should be created in separate specs and imported where needed.



# LISP CODE GENERATED FROM SPECS

The translation of executable specs to Lisp code is straightforward for the most part as Lisp is a higher-order functional language. Functional expressions go to lambda expressions and most Specware types are implemented as Lisp lists and vectors apart from the strings, numbers, characters and booleans which are implemented by the corresponding Lisp datatypes. This guide is meant primarily to help the user in calling and debugging the functions generated from a spec, so we concentrate on the translation of op names to Lisp names and the implementation of types. The implementation details of procedural constructs such as pattern-matching are omitted. The interested user is free to examine the Lisp code itself, which is simple but verbose for pattern-matching constructs.

## 7.1 Translation of Specware Names to Lisp Names

Specware ops are implemented using Lisp `defuns` if they are functions, `defparameters` otherwise. Their names are upper-cased and put in the package with the same name as the qualifier, or `SW-USER` if unqualified. However, if the name is that of a built-in Lisp symbol, the name is prepended with the character “!” and not upper-cased. If the qualifier of the op is the same as a built-in Lisp package then “-SPEC” is appended to the spec name to get the package name. For example, the Lisp code for the spec:

```
Z qualifying spec
  def two: Nat = 2
  def add1(x:Nat): Nat = x + 1
endspec
```

is

```
(DEFPACKAGE "Z")
(IN-PACKAGE "Z")

(DEFPARAMETER TWO 2)
(DEFUN ADD1 (X) (INTEGER-SPEC: :+-2 X 1))
```

## 7.2 Arity and Currying Normalization

All Specware functions are unary. Multiple argument functions are modeled using either functions with product domains, or curried functions. For efficiency we wish to exploit Common Lisp’s support of n-ary functions. Arity normalization aims to minimize unpacking and packing of products when passing arguments to functions with product domains, and currying normalization aims to minimize closure creation when calling curried functions. The saving is particularly important for recursive functions where there is saving at each recursive call, and in addition, currying normalization may permit the Common Lisp compiler to do tail recursion optimization. The naming scheme does not require knowledge of the definition of a function when generating calls to the function.

For each function whose argument is a product, two entry points are created: a unary function whose name is derived from the op as described above, and an n-ary function whose name has “-n” appended. For example, for

```
op min : Integer * Integer -> Integer
```

there are two Lisp functions, #'MIN-2 and #'|!min|. A call with an explicit product is translated to the n-ary version, otherwise the unary version is used. For example, min(1,2) translates to (MIN-2 1 2), and foldr min inf l translates to (FOLDER-1-1-1 #'|!min| INF L). When generating Lisp for a definition, the form is examined to see whether the definition is naturally n-ary. If it is, then the primary definition is n-ary and the unary function is defined in terms of the n-ary function, otherwise the situation is reversed. For example, given the definition

```
def min(x,y) = if x <= y then x else y
```

we get the two Common Lisp definitions:

```
(DEFUN MIN-2 (X Y) (if (<= x y) x y))
(DEFIN |!min| (X) (MIN-2 (CAR X) (CDR X)))
```

and given the definition

```
def multFG(x: Nat * Nat) = (F x) * (G x)
```

we get the two Common Lisp definitions:

```
(DEFUN MULTFG-2 (X Y) (MULTFG (CONS X Y)))
(DEFUN MULTFG (X) (* (F X) (G X)))
```

For each curried function (i.e. for each function whose codomain is a function) there is an additional uncurried version of the function with “-1” added  $n$  times to the name where  $n$  is the number of curried arguments. For example, for

```
op foldr: [key,a,b] (a * b -> b) -> b -> map(key,a) -> b
```

there are two Lisp functions, #'FOLDER and #'FOLDER-1-1-1.

As with arity normalization, the definition of a curried function is examined to see whether it should be used to generate the curried or the uncurried version, with the other being defined in terms of this primary version.

As well as producing more efficient code, the currying normalization makes code easier to debug using the Common Lisp trace facility. For example if a function has a call of the form foldr x y z, this call is implemented as (FOLDER-1-1-1 x y z), so you can trace FOLDER-1-1-1 to find out how it is being called and what it is returning.

## 7.3 Representation of Other Types

Character and String types are represented as Lisp characters and strings, Nat and Integer as Lisp integers, lists are represented using Lisp lists, and Bool true and false by the symbols T and NIL.

Sums are represented as the cons of the constructor name in keyword package and the fields of the constructor.

Binary products are implemented as cons cells (except for function arguments which are described in the previous section): CONS to construct and CAR and CDR to access the first and second fields. Non-binary products are implemented as vectors: constructed using VECTOR and the ith element accessed by (SVREF x i-1).

Records are implemented the same as products with the order of the fields being alphabetic in the field names.

Restrictions and comprehensions are implemented using their superset.

A quotient is represented as a vector of three elements: the quotient tag (which is the value of the Lisp variable SLANG-BUILT-IN ::QUOTIENT-TAG), the representation of the quotient relation, and the actual value in the underlying type.

# DEBUGGING GENERATED LISP FILES

## 8.1 Tracing

If you need to debug your application, there are a number of useful Lisp facilities you should be aware of. The simplest trick is to trace some functions you care about to see what they are doing.

```
(trace foo)
```

This will display the arguments to `foo` each time it is called, and will display the results each time it returns.

```
(untrace foo)
```

This will turn off any tracing on `foo`.

```
(untrace)
```

Untrace all functions.

## 8.2 Breaking

If you need a more detail view of runtime behavior, you might want to `BREAK` some functions you care about.

```
(trace foo :break-all t)
```

This will invoke the debugger each time `foo` is called, and upon each exit from `foo`.

Once you arrive in the debugger, you can use commands in the SLDB menu, and typing `h` will give a more comprehensive list of commands. Note, if you are in a break at the return of a function you can see the return value by evaluating `sb-debug:*trace-values*`.

SBCL's trace function is documented at <http://www.sbcl.org/manual/#Function-Tracing>.

## 8.3 Timing

If you are curious about the overall performance of your application, the `time` macro will provide some quick information:

```
(time (foo nil))
```

This will report the time and space used by `foo`, e.g.:

```
* (time (list 1 2 3))
```

Evaluation took:

```
0.000 seconds of real time
0.000006 seconds of total run time (0.000004 user, 0.000002 system)
100.00% CPU
2,481 processor cycles
0 bytes consed
```

```
(1 2 3)
```

Note that the number of bytes consed has the granularity of full pages, hence the 0 bytes consed in this small example. `time` is transparent, i.e., it returns whatever its argument would return, including multiple values, etc., so it is safe to intersperse it nearly anywhere.

Common Lisp has more facilities for rolling your own timers: see the generic Common Lisp documentation, or contact Kestrel Technologies.

## 8.4 Interrupting

Finally, note that a useful trick in Lisp is to start your application, e.g. `(foo nil)`, then at an appropriate time type `Ctrl-C` twice. This will interrupt your application and put you into the debugger where you can examine the state of the execution stack.



# EMACS USAGE IN SPECWARE

There are a number of emacs commands for Specware usage. We also list some useful general-purpose commands. There is a Specware mode for editing Specware files.

## 9.1 Specware Mode Commands

These are the commands available when editing `.sw` files. Some of them are available in the Specware menu in the menubar.

Key	Name	Description
tab	indent-line	Indents line based on context.
linefeed	newline-and-indent	Same as return followed by tab.
M-tab	back-to-outer-indent	Indents line for outer expression.
M-C-q	indent-sexp	Indents parenthesized expression following point.
M-C-`	indent-region	Indents region.
M-*	switch-to-lisp	Switch to *Specware Shell* buffer.
M-.	meta-point	Prompts for op or type name (with default based on symbol at point), and goes to its definition. Searches only the units that have been loaded. First searches for definitions visible from the current unit.
M-,	continue-meta-point	If previous meta-point command returned more than one definition, go to the next definition. Can be repeated.
M-	electric-pipe	Adds the skeleton of a new case to a case statement, properly indented.
C-c ;	comment-region	Comment out the region. With a negative argument uncomments the region.
C-c p	process-current-file	Does <code>proc</code> on current file. Also available in <code>dire</code> where it applies to the file on the current line. (This command will only work properly if the top-level directory for the file is in <code>SWPATH</code> ).
C-c C-p	process-unit	Prompts for <code>unitId</code> to process. Defaults to <code>unitId</code> for the current file.
C-c g	generate-lisp	Does <code>gen-lisp</code> on current file. With an argument compiles and loads the generated lisp.
C-c C-l	generate-lisp	Does <code>:swll</code> on current file (generates lisp for local definitions of current file and compiles and loads them).
C-c h	sw:convert-spec-to-haskell	Generates Haskell code for current spec.
C-c H	sw:convert-top-spec-to-haskell	Generates Haskell code for current spec as the top level.
C-c !	cd-current-directory	Does a <code>:cd</code> in the *Specware Shell* buffer to the directory of the current file. Also available in <code>dire</code> .

## 9.2 Specware Interaction Commands

These are the commands available in the \*Specware Shell\* buffer.

Key	Name	Description
M-.	meta-point	Prompts for op or type name (with default based on symbol at point), and goes to its definition. Searches only the units that have been loaded.
M-p	previous-input	Gets previous input.
M-r	previous-matching-input	Gets previous input matching regular expression (prompted for). Typically you can get a previous input by typing in a small substring. Repeating the command without changing the expression will find earlier matches.

## 9.3 Other Useful Emacs Commands

These are a few general Emacs commands which are useful when using Specware. Commands with no key sequences are executed using `M-x name`. XEmacs can be customized using the Options menu. For example, to make delete delete the selected region (as in most Word Processing Programs), mouse Options, Editing, Active Regions. To make the changes permanent, mouse Options, Save Options to Init File.

Key	Name	Description
C-h	help	Help options. C-h m gives help about the current mode. Also see Help menu in menubar.
M-/	dabbrev-expand	Does symbol completion based on nearby words in buffer. Repeated key presses find additional completions.
C-sh- middle	mode-motion-copy	Copies the (highlighted) identifier or expression under the mouse to point.
	igrep	Greps for string in files. Brings up buffer with matching lines. Mouse middle on a line to go to it. (In Windows requires installation of Cygwin.)
	igrep-find	Like igrep but searches in all subdirectories.
	fgrep	Like igrep except uses fgrep to search.
	fgrep-find	Like igrep-find except uses fgrep to search. all subdirectories.
C-x C-f	find-file	Prompts for file to edit. tab does filename completion.
C-x d	dired	Prompts for directory to edit. Note that commands process-current-file and cd-current-directory described in the previous section are available in dired mode.
	viper-mode	Vi emulation mode for people who like to edit using vi commands. Documentation is available under C-h i `.

## 9.4 X-Symbol Mode

X-Symbol is an Emacs package that allows the appearance of non-ascii symbols, such as mathematical operators and greek characters in file buffers. The non-ascii symbols have ascii representations that are stored in files, but converted to the non-ascii characters when the files are read into an emacs buffer with X-Symbol mode turned on. For example, the Specware symbol `\_forall` is displayed as `FIXMEUNICODEFORALL` and `\_or` is displayed as `||`. All the ascii representations begin with `"\_"`. X-Symbol mode can be turned on and off by Toggling X-Symbol under Options in the Specware menu. When it is on, there is an X-Symbol menu that provides several mechanisms for entering X-Symbols. For example, the characters may be chosen directly from the menu from different categories, such as `||` under the Operators sub-menu and `FIXMEUNICODEFORALL` under the Symbol sub-menu. This sub-menu also shows keyboard commands for the symbol. E.g., the keyboard shortcut for `||` is `-- /1`. Alternatively, you can use the command `c-`, after `/`. Repeated `c-`, commands will get you related X-Symbols. You can see all the available X-Symbols by selecting GRID of Characters under Other Commands in the X-Symbol menu.

## 9.5 Hide/Show Commands

There is experimental support for hiding bodies of specs, definitions and other multi-line constructs. The main command for hiding is `Shift-Middle-Click`, i.e., holding down the shift key and middle-clicking near the beginning of a multi-line construct. This will hide all but the first line of the construct and display a `...` at the end of the line to indicate that the body is hidden. Doing a shift- middle-click on this line will expand the body. Thus shift-middle-click acts as a hiding toggle.

There is a Hide/Show menu. The most useful commands it contains are `Show All` and `Hide All`. `Show All` removes all hiding in the file. `Hide All` hides the bodies of all top-level constructs. I.e., if it is multi-unit file, the

the bodies of the units will be hidden, and if is a single-unit file then the bodies of multi-line declarations such as definitions and axioms will be hidden.

# INDEX

## A

absolute unit identifier, *see* resolution

## C

cd  
    shell-command, 15  
cf  
    shell-command, 28  
cinit  
    shell-command, 17  
cl  
    shell-command, 28  
ctext  
    shell-command, 18

## D

dir  
    shell-command, 15  
dirr  
    shell-command, 15

## E

e  
    shell-command, 18  
eval  
    shell-command, 18  
eval-lisp  
    shell-command, 18  
exit  
    shell-command, 28

## G

gen-c  
    shell-command, 22  
gen-haskell  
    shell-command, 25  
gen-haskell-top  
    shell-command, 25  
gen-java  
    shell-command, 19  
gen-lisp

    shell-command, 19

## H

help  
    shell-command, 15

## L

ld  
    shell-command, 28  
lgen-lisp  
    shell-command, 19

## M

make  
    shell-command, 23

## O

obligations  
    shell-command, 12, 17  
oblis  
    shell-command, 17

## P

p  
    shell-command, 16  
path  
    shell-command, 15  
pragma  
    header, 28  
    instance, 27  
    morphism, 26  
    strict, 27  
    translate, 25  
proc  
    shell-command, 16  
prove  
    shell-command, 16  
punits  
    shell-command, 17

## R

relative unit identifier, *see* resolution

resolution

unit identifier, 12, 13

## S

shell-command

cd, 15

cf, 28

change directory, 15

cinit, 17

cl, 28

ctext, 18

dir, 15

dirr, 15

e, 18

eval, 18

eval-lisp, 18

exit, 28

gen-c, 22

gen-haskell, 25

gen-haskell-top, 25

gen-java, 19

gen-lisp, 19

help, 15

ld, 28

lgen-lisp, 19

make, 23

obligations, 12, 17

oblgs, 17

p, 16

path, 15

proc, 16

prove, 16

punits, 17

show, 17

showx, 17

show

shell-command, 17

showx

shell-command, 17

## U

unit identifier

resolution, 12, 13