Knit User's Manual and Tutorial
Version 1.0.0

Flux Research Group
School of Computing
University of Utah

http://www.cs.utah.edu/flux/

February 2001

Contents

1 Introduction **1**

2 Compiling and Installing Knit **3**
- 2.1 Acquiring Knit . . . 3
- 2.2 Building the Knit Programs . . . 4
  - 2.2.1 Compiling `libelf` . . . 5
  - 2.2.2 Configuring Knit . . . 5
  - 2.2.3 Compiling Knit . . . 6
- 2.3 Installing Knit . . . 6
- 2.4 Testing Knit . . . 7
- 2.5 Related Software . . . 7
  - 2.5.1 The OSKit . . . 7
  - 2.5.2 `gcc` for Knitted Code . . . 7

3 Using Knit **9**
- 3.1 The `knit` Compiler . . . 9
  - 3.1.1 Output Files . . . 9
  - 3.1.2 Command Line Options . . . 10
  - 3.1.3 The `knit_generated.mk` File . . . 12
- 3.2 The `knitdoc` Documentation Generator . . . 12
- 3.3 The `mk_unit` Template Generator . . . 13
- 3.4 The `rename_dot_o_files` Object Editor . . . 16

4 Tutorial **17**
- 4.1 Unit Basics: The `hello` Example . . . 17
  - 4.1.1 The Unit Model of Software Components . . . 17
  - 4.1.2 A Unit File . . . 18
  - 4.1.3 Imports and Exports . . . 19
  - 4.1.4 Dependencies . . . 19
  - 4.1.5 Files . . . 20
  - 4.1.6 Compiling the `hello` Program . . . 20
- 4.2 Using Multiple Units: The `msg` Example . . . 22
  - 4.2.1 Bundletypes . . . 22
  - 4.2.2 Renaming . . . 23
  - 4.2.3 Compound Units . . . 24
  - 4.2.4 Compiling the `msg` Program . . . 25
- 4.3 Knitting Tricks: The `calc` Example . . . 26

4.3.1 Initializers and Finalizers 27
4.3.2 More About Dependencies 28
4.3.3 More About Renaming 29
4.3.4 Wrappers and Transparent Interposition 30
4.3.5 Multiple Instantiation 32
4.3.6 Summary: The `Calc` and `Calc_Counted` Units 33
4.3.7 Compiling the `calc` Program 34
4.3.8 Optimizing the Code via "Flattening" 35
4.4 Other Knit Features 36

5 Debugging Knitted Programs **37**

6 Knitting the OSKit **39**

Chapter 1

Introduction

Knit is a component definition and linking language which can be used with little or no modification with C and assembly code. Knit supports component hierarchies, cyclic component dependencies, automatic scheduling of initializers and finalizers, an extensible constraint system to detect errors in component composition, and cross-module inlining to largely eliminate the overheads of componentization.

The goal of this document is to describe the things you need to know in order to start using Knit, including:

- How to install Knit on your system (Chapter 2).
- How to invoke the Knit compiler (Section 3.1).
- Other Knit support tools including a tool for generating HTML documentation (Section 3.2) and a tool for generating unit descriptions (Section 3.3).
- A tutorial on how to use Knit and how to use it effectively (Chapter 4).

The complete Knit language is described in a separate document, *Report on the Language Knit: A Component Definition and Linking Language*. **You should read this user's manual, especially the tutorial chapter, before reading the more detailed Knit language report.**

Chapter 2

Compiling and Installing Knit

The `INSTALL` file in the topmost directory of the Knit source tree contains an abbreviated version of these instructions.

Knit consists of the following programs:

`knit`	The Knit unit language compiler. This is the heart of the system and the program that one runs to process a '`.unit`' file.
`knitdoc`	A program that produces HTML documentation from '`.unit`' files.
`mk_unit`	A Perl script that can produce a skeleton '`.unit`' file from a set of object files. This script is useful for "importing" existing code into Knit.
`rename_dot_o_files`	A tool used by `knit` to transform '`.o`' files. While normally invoked "behind the scenes," this program can also be used on its own to edit symbols that appear within an object file.
`knit_c_parser`	A parser for C source code, invoked internally by `knit`.
`knit_smartmv`	A small Perl script for moving files, invoked internally by `knit`.
`knitGenBundles`	A program that produces bundle declarations, invoked internally by `mk_unit`.

`knit`, `knitdoc`, and `knitGenBundles` are written in the Haskell programming language, and the helper programs `rename_dot_o_files` and `knit_c_parser` are written in C. Because Knit can be somewhat tricky to compile, most users will want to download the Knit tools in precompiled form. Users who want to work on Knit itself, however, will need to be able to compile Knit from source.

FreeBSD and Linux are the current Knit development platforms. If you build Knit on an interesting platform, the Knit developers would love to know about it. Of course, if the build fails, they would like to know about that, too! Send build reports, bug reports, praise and damnation to `knit-users@flux.cs.utah.edu`.

2.1 Acquiring Knit

Knit is available on the World Wide Web under `http://www.cs.utah.edu/flux/knit/`. Currently, Knit is distributed in two forms: as precompiled binaries for a variety of operating systems, and as source code. We strongly recommend that you use the precompiled binaries for your platform if available. (Please contact the Knit authors if precompiled binaries for your platform are not available. If demand warrants, your platform may be added to the supported set.)

2.2 Building the Knit Programs

If you are using a binary distribution of Knit, you should skip the instructions below and proceed to Section 2.3.

In order to build Knit from source, you will need:

An ELF-based system	Knit is currently implemented only for ELF-based systems.
A Haskell compiler	Version 4.08.1 of `ghc`, the Glasgow Haskell Compiler[1], is our usual compiler. Instead of a compiler, one can use a Haskell interpreter such as Hugs[2], but this is definitely not recommended for large projects.
`green-card`	`green-card`[3] is a foreign function interface generator for Haskell. Green Card is required when one uses `ghc` to compile Knit, but is optional if one uses Hugs.
An ANSI C compiler	`gcc` 2.95.2 is our usual compiler. We strongly recommend `gcc` for building Knit. As of this writing, some of the Knit example programs make use of `gcc` language extensions, and the OSKit (described in Section 2.5.1 requires `gcc` in any case. Additionally, as described in Section 2.5.2, advanced projects may want to build a special version of `gcc` for optimized Knit-based code.
GNU `make`	Version 3.77 has been tested.
`flex` or `lex`	`flex` version 2.5.4 has been tested. Vendor-provided versions of `lex` should work, but this has not been extensively tested.
`bison` or `yacc`	`bison` version 1.25 has been tested. Vendor-provided versions of `yacc` should work, but have not been extensively tested.
`libelf`	Version 0.7.0 has been tested. `libelf`[4] is a freely available library for manipulating ELF object files. `libelf` is preinstalled on some operating systems, such as Linux. For other systems, `libelf` is available from the World Wide Web. *Note the special instructions for building `libelf` in Section 2.2.1.*
Disk space	You will need about 10 MB of free disk space to build Knit and the current set of Knit example programs.

Once you have the above-listed tools, you will need to (1) unpack the Knit source distribution `tar` file and (2) create a build tree to contain the object files. Assuming that you did these things in your home directory, your directory setup would look something like this:

[1]http://www.haskell.org/ghc/
[2]http://www.haskell.org/hugs/
[3]http://www.dcs.gla.ac.uk/fp/software/green-card/
[4]http://www.stud.uni-hannover.de/ michael/software/

`~/knit-1.0.0/`	The Knit source tree, unpacked from `knit-1.0.0.src.tar.gz`.
`~/obj/`	The directory you create to be the root of your Knit object tree. (Of course, you can call this directory anything you like. It does not have to be called `obj`.)

2.2.1 Compiling `libelf`

Before configuring or compiling Knit, you must first have the `libelf` library and headers available on your system. If your operating system has `libelf` preinstalled, great: you are done with this step. However, if your system does not come with `libelf`, you must download and compile (and optionally install) `libelf` for yourself before proceeding. See Section 2.2 for download and version information.

> **WARNING:** If you have problems with the Knit `rename_dot_o_files` program — i.e., if it seems to produce invalid object files — you may need to build `libelf` so that it does not use the `mmap` system call to manipulate files. (The Knit authors have seen this problem occur when the object file is located in an NFS-mounted filesystem.) To build `libelf` so that it does not use `mmap`, first configure the library normally, and then comment out the `HAVE_MMAP` definition in the generated `config.h` file. Finally, build and install the library.

If you do not have `libelf` installed on your system, you will need to use the `--with-libelf` command line option when you configure Knit for your system, as described in the next section.

2.2.2 Configuring Knit

Given the directory setup described in Section 2.2, you would type the following commands to configure Knit for your operating system:

```
cd ~/obj
../knit-1.0.0/configure
```

The `configure` script in the root of the Knit source tree is an ordinary `configure` script, generated by GNU `autoconf`. Type "`configure --help`" to see all of the script's command line options. In addition to all of the standard options accepted by `autoconf`-generated scripts, Knit's `configure` script accepts the following Knit-specific options:

`--with-libelf=`*`DIR`*	Use the `libelf` library installed in *`DIR`*. The Knit Makefiles will look for the `libelf` header files in *`DIR`*`/include/`, and will look for the `libelf` library itself in *`DIR`*`/lib/`.
`--with-greencard-dir=`*`DIR`*	Use the Green Card files in *`DIR`* to build certain Knit C source code files. This option is useful for Hugs-based builds only.

Additionally, the Knit configure script examines the values of the following environment variables:

`HC`	The name of the Haskell compiler to be used. If this variable is unset, the default value `ghc` is used.
`GREENCARD`	The name of the the Green Card program. Default: `green-card`.
`RUNHUGS`	The name of the Hugs program. Default: `runhugs`.

2.2.3 Compiling Knit

Finally, you are ready to compile the Knit tools! If you are using the Glasgow Haskell Compiler, type the following commands to build Knit and its auxiliary programs:

```
cd ~/obj/compiler/ghc
make all
```

Assuming all goes well, these commands should produce all of the Knit programs that need to be compiled.

If you are using Hugs, type the following instead:

```
cd ~/obj/compiler/hugs
make all
```

These commands will produce some object files and the C-based helper programs only (i.e., `knit_c_parser` and `rename_dot_o_files`). Since Hugs is an interpreter, there is no need to compile the Haskell-based programs.

Of course, if you are using a precompiled binary release of Knit, you do not need to build the binaries at all: you already have them! But you still have to *install* the programs — read on.

2.3 Installing Knit

You can install the Knit compiler programs in a "standard" place on your system by typing "`make install`" in the `bin` subdirectory of your Knit build tree (in our running example, `~/obj/bin`).

Special instructions for binary distributions: If you are using a binary distribution of Knit, you will nevertheless need to run the Knit `configure` script, which will (1) tailor certain Knit scripts to your system, (2) configure the installation process for your system, and (3) produce the Knit example program Makefiles. We encourage you to run the `configure` script in a directory that is separate from your Knit distribution tree, because Knit will produce many files when processing the examples. So, to configure and install a binary distribution, type something like the following:

```
mkdir ~/obj
cd ~/obj
../knit-1.0.0/configure
cd bin
make install
```

Whether you use a source or binary distribution, by default, the root installation directory is `/usr/local`. You can change this default by specifying the `--prefix` option to `configure`, e.g.:

```
../knit-1.0.0/configure --prefix=/usr/local/knit-1.0.0

make -C ~/obj/compiler/ghc all     # Build Knit
make -C ~/obj/bin          install # Install under '/usr/local/knit-1.0.0'
```

It is not necessary to install Knit in order to use the provided example programs (described in the next section) or to build your own Knit-based programs.

2.4 Testing Knit

Knit comes with a small set of example programs; these are located in the `examples` subdirectory of the distribution source tree. Running your newly compiled Knit tools on the provided examples is easy. In a sentence: go to the `examples` subdirectory of your Knit object tree and type `make`. This will build all of examples. If everything builds, congratulations! You are now a "techknitian"!

The various example programs are described in greater detail in Chapter 4.

2.5 Related Software

The software described below is not part of Knit, but is often used in conjunction with Knit.

2.5.1 The OSKit

For projects involving operating systems or similar low-level software, you may want to use Knit in conjunction with the OSKit. The OSKit is a framework and set of modularized components and library code, together with extensive documentation, for the construction of operating system kernels, servers, and other OS-level tools. Its purpose is to provide much of the infrastructure "grunge" that usually takes up a large percentage of development time in any operating system or OS-related project, and allow developers to concentrate their efforts on the unique and interesting aspects of the new OS in question. The OSKit is available on the World Wide Web under `http://www.cs.utah.edu/flux/oskit/`.

Recent versions of the OSKit can be configured to work with Knit. Instructions for using Knit with the OSKit are in Chapter 6. Knit-specific files in the OSKit are located in a subdirectory called `oskit/knit`. Look there for unit definition files and related files. Refer to the OSKit documentation for further information about the OSKit.

> **WARNING:** Because Knit and the OSKit are both active and evolving projects, you must be careful to get a "matched set" in order to use them together! Knit version 1.0.0 is intended to work with OSKit version 20010214 (a snapshot release). Later versions of either system may not work with previous versions of the other — *backward compatibility is not guaranteed!*

2.5.2 `gcc` for Knitted Code

If you use Knit's "flattening" optimization on certain C sources, including the OSKit, you may need to use a special version of `gcc` in order to compile your Knit-generated C code. This special version of `gcc` must ignore certain type errors that may be introduced by Knit, because Knit does not unify identical type declarations that originate in two separate C files. This sounds really dodgy, but we are assuming that you have already built your system without flattening using a normal C compiler. If the unflattened version is free of type errors, then compiling with flattening and the hacked compiler should work just fine.

The instructions for producing a patched version of `gcc` are contained in the file `unsupported/gcc.patch` in the Knit distribution tree. The effect of the patch is to allow assignments to variables when the assigned value has a different type, so long as the types of the variable and value have the same storage size. This patch relaxes the C type rules and is okay as long as your program can be compiled in a normal way by an unpatched version of `gcc`.

> **WARNING:** If you make a patched version of `gcc`, you should be careful to install it in a very special place with a very special name, so that nobody will use it accidentally to compile anything other than Knit-generated code.

Chapter 3

Using Knit

Knit operates on *unit files*: files that describe programs and program parts in terms of their of software components. Each component is called a *unit*. Unit descriptions are written in a textual specification or “programming” language and stored in unit files (files whose names end with ‘`.unit`’), which may be processed by Knit.

This chapter describes the Knit tools that operate on unit files, that produce unit files, or that may otherwise by useful to Knit users. Chapter 4 is a tutorial on the unit description language, i.e., the contents of unit files.

3.1 The `knit` Compiler

The Knit unit file compiler is simply called `knit`. The command line syntax is as follows:

```
knit [option ...] [var=value ...] unit-file unit-name
```

where *`unit-file`* is the name of the unit file to be read, and *`unit-name`* is the name of the “topmost” unit that should be processed, i.e., the unit that describes the complete program or library that is to be compiled. (The other command line arguments will be described in a moment.)

3.1.1 Output Files

Assuming that compilation is successful, `knit` produces a set of output files:

`knit_generated.mk`	`knit` produces a set of `make` rules for building the archive (‘`.a`’) files that will contain the compiled code for the unit named on the `knit` command line. The `knit_generated.mk` file is not a complete Makefile; rather, it is designed to be included from another Makefile.
`makefile`	If the `MAKEFILE=`*`template`* argument was specified on the command line (as described below), `knit` will use the specified *`template`* to create a `makefile` in the current directory. (Presumably, `makefile` will include the `knit_generated.mk` file.) Currently, the template file is simply copied; no transformations are made on the file contents.

`rename_*`	These files are inputs to the `rename_dot_o_files` tool (Section 3.4) and describe how symbols in generated object (‘`.o`’) files should be renamed as part of implementing the unit. The `rename_*` files are referenced by the rules in `knit_generated.mk`; you do not need to deal with these files directly.
`knit_inits.c`	This file contains the implementations of the generated `knit_init` and `knit_fini` C functions. These functions will be called to run your units’ initializers and finalizers, respectively. (Initializers and finalizers are described in Section 4.3.1.)
`*_anon_*.c`	If your unit definitions contain literal C code, `knit` will move this “anonymous” code into C source files so that it may be compiled. These files may also be made if you use Knit’s “flattening” optimization (described in Section 4.3.8).

`knit` may also leave certain temporary files behind, with names like `TMP`, `*xxx`, and `*yyy`. You may safely ignore these files.

3.1.2 Command Line Options

The command line options to `knit` are as follows:

`-X`	Do not create any output files. This option is useful when you simply want to check the correctness of your unit specifications.
`-f`	Perform the code “flattening” optimization described in Section 4.3.8. In brief, this option tells Knit to collect and weave all of your C code into one file, so that it may be better optimized by your C compiler. Knit does not perform this optimization by default.
`-c`	Check the constraints that are specified in the unit specifications. Constraints and constraint-checking are described in the the *Report on the Language Knit: A Component Definition and Linking Language* (in the `doc/report` directory of the Knit distribution). By default, Knit does not enforce constraints.

In addition to these options, `knit` looks for command line arguments of the form *`var=value`*. The compiler processes these arguments in three ways.

First, all of the variable definitions are copied into the output `knit_generated.mk` file. This provides a convenient way for you to specify certain `make` variables at the time Knit is run. Moreover, it makes it possible for you to put references to these variables at certain points in the *unit file itself* — for example, in the specifications of C file names. These references will be expanded when your C code is actually compiled by `make`.

Second, *`var=value`* arguments to `knit` are put into the environment of any subprocesses created by the compiler. This is principally useful when `knit`’s code flattening optimization is enabled; with flattening, `knit` uses commands like the following to preprocess your unit code:

```
env var=value ... \
    sh -c 'gcc -P -E $KNIT_CPPFLAGS ... source-file.c'
```

Because the variables are put into the environment, one can use variable references within **directory** specifications in a unit file. In fact, it is very good practice to use variables in **directory** specifications, because this makes your unit files less dependent on the exact organization of your source files.

Third and finally, certain variables have special meaning to `knit` itself. These variables are:

`UNIT_PATH=`*`dirs`* *(Required.)*	Specify the search path for input unit files. The *`dirs`* path is a colon-separated list of directory names. *Note that `knit` requires that you provide a value for `UNIT_PATH` on the command line.*
`MAKEFILE=`*`file`*	If specified, `knit` will use the given *`file`* as a template for creating a `makefile` in the current directory. If `MAKEFILE` is not specified on the command line, `knit` will not create a Makefile for you. (In any case, `knit` *will* create the `knit_generated.mk` rule file.)
`KNIT_TOOLS=`*`dir`*	Tell `knit` to find its auxiliary programs (e.g., `knit_c_parser` and `knit_smartmv`) in the specified directory. This option is primarily used when invoking a non-installed set of Knit tools, such as the programs within a Knit build tree. If `KNIT_TOOLS` is not specified on the command line, `knit` will look for its helper programs along the user's usual program search path.
`KNIT_CPPFLAGS=`*`cppflags`*	When flattening is enabled, `KNIT_CPPFLAGS` contains any extra flags that `knit` should pass to the C preprocessor. The default is to pass no extra arguments. When flattening is not enabled, this variable has no effect.
`KNIT_BUDGET=`*`number`*	When flattening is enabled, `KNIT_BUDGET` provides control over the amount of inlining that should be performed. *Very roughly*, the value of `KNIT_BUDGET` is the total number of static RISC instructions that should be spent on or saved by inlining. Positive values of *`number`* represent spending (i.e., increased code size) while negative values represent saving (reduced code size). Code size can be reduced by removing dead functions, by inlining functions that are only used once (which eliminates instructions to push arguments, call the function, and return), and by inlining trivial functions whose body requires fewer instructions than a function call. Any such savings count toward achieving the overall budget: increasing the number of inlined instructions that `knit` may put elsewhere. If unspecified, the default budget is `0`. In practice, the value of `KNIT_BUDGET` is only very loosely correlated with the size of the final binary. This is because Knit operates on the C source code, and therefore has only indirect control over the optimizations that may (or may not) be performed by the C compiler. Beyond simple inlining and dead function elimination, Knit does not try to predict the effect of other optimizations that the C compiler may provide.

Note that `knit` processes variable settings *from the `knit` command line only*. In particular, `knit` does not look for settings of environment variables. (You do not want your complete environment copied into the `knit_generated.mk` file, do you?)

3.1.3 The `knit_generated.mk` File

Most of the recipe for building your program or library is contained in the `knit_generated.mk` file, which is produced for you by `knit`. As described previously, this file contains the `make` rules for (1) compiling the necessary source files into object files, (2) manipulating the object files as required to make the proper cross-unit connections, and (3) combining the resultant files into one or more archive files. This is as far as the `knit_generated.mk` rules go, however. More is needed in order to finish the job of making a complete, final program.

Since the rules for "finishing the job" are not known to Knit, Knit is designed to make it easy for you to write your own Makefile containing the necessary rules. The idea is that your Makefile will include the `knit_generated.mk` rules file, and then provide the higher-level rules for the final assembly of your program. Typical rules for final assembly might look like this:

```
$(PROGRAM): knit_inits.o $(KNIT_LIBS) ...
        $(CC) -o $@ --begin-group $^ --end-group

knit_inits.o: knit_inits.c
        $(CC) -c $< $(CFLAGS)
```

There are four important things to notice about the above rules. First, the complete program is made by linking together the compiled `knit_inits.o` file and all of the libraries that contain your program's unit code. If your program requires non-Knitted objects in addition to the Knit-generated libraries, these would also be listed in the rule. Second, the value of `KNIT_LIBS` is set in the `knit_generated.mk` file. That file defines a variable `KNIT_OBJS` as well, and any other variables that were specified on the `knit` compiler command line, as described previously in Section 3.1.2. Third, the set of program objects is given to the C compiler as a group, between the `--begin-group` and `--end-group` options. This idiom — which is specific to `gcc`, unfortunately — eliminates potential problems that the linker might have in resolving symbols. Finally, the Knit-generated `knit_inits.c` file is not in a unit, and therefore must be compiled and linked into your program explicitly. Think of the code in `knit_inits.c` as part of the "runtime environment" for your Knitted code.

The example programs that come with Knit (located within the `examples` subdirectory of the software distribution) each have a complete Makefile that you can easily copy and adapt for your own work.

3.2 The `knitdoc` Documentation Generator

> **WARNING:** The `knitdoc` program does not currently work if you build Knit with Hugs. `knitdoc` requires some Haskell libraries that are provided with `ghc`, but not with Hugs.

The `knitdoc` program produces HTML-format documentation from a unit file. The command line syntax is:

```
knitdoc [var=value ...] dest-dir unit-files
```

The command line arguments are:

var=value	Variable settings, as described previously for `knit` in Section 3.1.2. Like `knit`, `knitdoc` requires that the `UNIT_PATH` be specified on the command line.
dest-dir	The name of the directory into which the HTML output files will go. Note that this directory must already exist.

`unit-files`	The names of the unit files to be processed. Any files that are included by *`unit-files`* will be processed as well.

The output of `knitdoc` is a set of HTML files describing the units and bundletypes that are defined in the unit files. (Other kinds of top-level definitions are not yet translated.) The "root" of the documentation is found in the generated `index.html` file.

Although the generated HTML is determined almost entirely by the unit and bundletype declarations themselves, `knitdoc` supports *documentation comments* (also called "*doc comments*") that are similar to those found in Java. In a unit file, a doc comment begins with the three-character sequence "`/*#`" and ends with the sequence "`#*/`". Every character between these delimiters is part of the comment; leading asterisks and whitespace are not discarded as they would be in a Java doc comment.

```
/*# This comment describes the ‘Part’ unit... #*/
unit Part = {
  ...
}
```

A doc comment that precedes a unit or bundletype definition will be copied verbatim into the generated HTML page for the definition. Therefore, the body of a doc comment should be written as valid HTML.[1] Documentation comments must *precede* the unit or bundletype definition; they cannot be used to document *parts* of a definition. Also, note that `knitdoc` does not currently support Java-style *tagged paragraphs* within doc comments (e.g., paragraphs marked with tags like `@see` or `@author`).

3.3 The `mk_unit` Template Generator

The `knit` compiler and `knitdoc` documentation generator both work on unit files, and ultimately, a unit file must be written by a person who understands the purpose and structure of the unit-encapsulated C code. To ease the task of writing a unit file, however, the Knit tool suite includes `mk_unit`, a small script that can aid the programmer by producing much of the unit file "boilerplate."

`mk_unit` reads a set of object (‘`.o`’) files, analyzes the imported and exported symbols, heuristically groups related symbols into bundles, and finally outputs (to stdout) the boilerplate for a unit that can encapsulate the analyzed objects. The command line syntax of `mk_unit` is:

```
mk_unit [-n name] object-files [-- other-object-files [-- genbundle-args] ]
```

where the options and arguments are as follows:

`-n` *`name`*	Use *`name`* as the name of the generated unit description. Without this option, `mk_unit` gives the generated unit a dummy name.
`object-files`	The names of the object files to be processed. The `mk_unit` script creates one unit definition that describes the collection of objects, not one unit description for each object. If you want each object as a separate unit, simply run `mk_unit` separately on each.

[1]Although you are free to use HTML in doc comments, it would be wise to do so sparingly. Future versions of `knitdoc` may support multiple output formats (e.g., $\LaTeX$) in addition to HTML.

`--` *`other-object-files`*

The names of object files that may import symbols from the unit being generated. When dividing an existing program or library into multiple units, it is useful for `mk_unit` to know which functions and variables are actually used across unit boundaries. By knowing this, `mk_unit` can make a better unit definitions, ones in which the exports are driven by actual cross-unit connections.

Thus, `mk_unit` needs information about the "environment" for the unit being generated, and this is given by *`other-object-files`*. These files are used to separate the set of exported symbols into those that *must be exported* from the unit being generated and those that *one may choose not to export* from the unit.

If no information about the "unit environment" is available, one can simply specify an empty set of *`other-object-files`*.

`--` *`genbundle-args`*

If a second `--` appears on the command line, all remaining arguments are passed through to the `knitGenBundles` program. This program is invoked by `mk_unit` to sort the imported and exported symbols into related groups — what Knit calls *bundles*. The arguments that may usefully appear after `--` are the following:

`UNIT_PATH=`*`dirs`* *(Required.)*	Search path for unit files, as described in Section 3.1.2.
`var=value`	Other bindings as described in Section 3.1.2.
`unit-file`	The file from which to read bundletypes.

The `mk_unit` script uses the **bundletype** definitions in the given *`unit-file`* to organize the import and export symbols of the unit definition being created. By providing the set of bundletypes being used in your project, you can *greatly* improve the quality of the unit definitions generated by `mk_unit`.

Note that if a second `--` option is not given to `mk_unit`, or if there are no *`genbundle-args`* on the command line, then `mk_unit` will not invoke `knitGenBundles` to group symbols. Instead, `mk_unit` will produce a unit definition that has a single import bundle and one or two export bundles. (There may be two export bundles if a non-empty set of *`other-unit-files`* was specified.)

The output of `mk_unit` is a unit definition of the following form:

```
unit name = {
  imports[ ... ];
  exports[ ... ];
  depends{ exports + inits + finis needs imports; };
  files{ object-files } with flags {};
}
```

The following transcript shows how `mk_unit` could be used to generate a unit definition for one of the example programs that comes with Knit. In the Knit distribution, the file `examples/calc/main.c` contains the main function for a calculator-like program called `calc`. (See Section 4.3.) Since `calc` is a Knit example, the file `examples/calc/calc.unit` already contains a unit definition for the code in `main.c`. Nevertheless, we can use `mk_unit` to generate a new unit definition for the code. We might do this in order to check the hand-written unit, for example.

```
cd examples/calc
make main.o
mk_unit -n Main main.o -- -- calc.unit
```

`mk_unit` processes `main.o`, reads the **bundletype** definitions from the `calc.unit` file, and finally outputs the following unit definition:

```
unit Main = {
  imports[ Repl_T : Repl_T, /* {repl} */
         ];
  exports[ Main_T : Main_T, /* {main} */
         ];
  depends{ exports + inits + finis needs imports };
  files{
    "main.o",
  } with flags {};
}
```

`Repl_T` and `Main_T` are the names of bundletypes defined in the `calc.unit` file. If you compare the above output to the actual definition of `Main` in `calc.unit`, you will see that the `mk_unit`-generated definition and the actual definition are nearly identical. `mk_unit` did not just copy the `Main` definition from `calc.unit`, though — it analyzed `main.o` and produced its own unit definition!

While a `mk_unit`- generated unit definition will be "complete," it will almost certainly need some hand-tweaking in order to be most useful. For instance, you may want to:

- change the organization of the imports and exports,
- reclassify an exported common symbol as an imported common symbol,
- specify linking constraints,
- specify initializers and finalizers,
- provide finer-grain dependency information,
- change the **files** list to refer to the source C files, or
- specify C preprocessor flags for the files.

Most of these Knit language features are described in Chapter 4.

Do not be concerned that you will need to edit the generated unit file. The purpose of `mk_unit` is to "get you off the ground," not to create the final unit definitions for your project. The `mk_unit` script is something that you are expected to run once for each set of objects in your project, and then never again.

Finally, note that because `mk_unit` is written is Perl, you can easily modify the script to suit the needs of specific projects.

3.4 The `rename_dot_o_files` Object Editor

The final Knit tool described in this chapter is `rename_dot_o_files`, the object file editor that is invoked by Knit-generated Makefiles. While you would never need to invoke `rename_dot_o_files` by hand in the normal course of using Knit, you might find `rename_dot_o_files` to be of use in other projects. So, for hackers and the curious, we describe the program here.

The basic purpose of `rename_dot_o_files` is to change the names of non-local symbols (i.e., import and exports) that appear within an object file. Specific symbol renamings are described in a *renaming file*. Symbols not listed in the renaming file are renamed by applying a prefix, which is specified on the command line. The command line syntax of `rename_dot_o_files` is:

```
rename_dot_o_files prefix rename-file object-file
```

where the arguments are as follows:

prefix	The prefix that should be applied by default to every non-local symbol in the object file. However, if an explicit renaming pattern is given for a symbol in the renaming file, then the prefix is *not* applied to that symbol.
rename-file	The name of the renaming file. This file contains a set of renaming specifications, each on a separate line, and each of the form "*from*=*to*" where *from* and *to* are symbols. In the object file being edited, every occurrence of the (non-local) symbol *from* will be replaced with the symbol *to*. `rename_dot_o_files` is somewhat fussy about the format of this file and the file should not contain any unnecessary white space.
object-file	The name of the object file to be edited. Note that the object file is edited "in place": `rename_dot_o_files` mutates the given file instead of producing a new object file.

You might look at the files produced by `knit`, as described in Section 3.1.1, to get a better feel for how `rename_dot_o_files` can be used.

Chapter 4

Tutorial

This chapter presents the Knit unit language by taking you through a series of example programs. Complete source code for each of the examples is included in the Knit software distribution, within the `examples` subdirectory, so you can try out Knit as you read this chapter. The examples used in this chapter include:

`examples/hello`	The standard "Hello, World!" program, demonstrating the simplest possible use of Knit.
`examples/msg`	A step up from the previous example, showing how one can build a program by combining multiple units in a flexible manner.
`examples/calc`	A four-function calculator program built from several units, with initialization, finalization, and program instrumentation woven by Knit.

Note that this tutorial does not (yet!) cover all of the features in the Knit unit language. For a full and formal treatment of the Knit language, please read the *Report on the Language Knit: A Component Definition and Linking Language*, which is also part of the Knit software distribution.

4.1 Unit Basics: The `hello` Example

Our first example has two goals: first, to introduce a few basic Knit concepts such as units and bundles, and second, to take you through the steps of compiling a program with Knit. So, to get started, we will show how to create the standard "Hello, World!" program with Knit.

4.1.1 The Unit Model of Software Components

In Knit, a program is made up of software components called *units*. A unit is a "logical" wrapper around code, and by "logical" we mean two things:

1. First, units are *compile-time* or *configure-time* components, not *run-time* components. In contrast to technologies such as COM and CORBA, unit boundaries are "compiled away" when your program is turned into an executable. In this way, units are "logical" wrappers, in that they are not present when your code is ultimately run.

2. Second, units are defined in one or more *unit files* that are separate from the files containing your program's C code or other source code. This makes it easier to use legacy C code with Knit: the

units and their implementations are logically separate. You do not have to insert unit definitions or otherwise modify your C files in order to use Knit.

In other words, a Knit unit is a kind of description of the code that it "wraps." This description is used when your program is *compiled*, not when your program is executed. Therefore, a unit describes the things that Knit must know about a piece of code in order for that code to be combined and linked with other units to form a complete system. These things include:

- *The basic interfaces of the code.* What functions and variables are defined by the unit? What functions and variables must be provided to the unit? In Knit terminology, these are the *imports* and *exports* of a unit, respectively.
- *How is the unit implemented?* A unit can be implemented by code from one or more separate files: Knit can currently work with C files, assembly files, and object (`.o`) files. A unit can also be created in a hierarchical fashion by composing and wiring together other units.

  If a unit is implemented by code from a set of files, we say that the unit is *atomic*. In contrast, if a unit is implemented by a set of other units, we say that the unit is *compound*.
- *What constraints exist on the use of this unit?* Some units must be initialized before normal use. Other units may not require initialization themselves, but may import functions from other units, and those "imported units" may need to be initialized.

4.1.2 A Unit File

The following unit definition shows how the above-described features of a unit are expressed in Knit. The next few sections will discuss the parts of this unit definition in more detail.

```
unit Hello = {
    imports [ io: {printf} ];
    exports [ main: {main} ];
    depends { main needs io; };
    files { "hello.c" };
}
```

This definition comes from the file `examples/hello/hello.unit`. As you can see, Knit unit definitions are written in a textual specification or "programming" language. These definitions are grouped and stored in *unit files* (with names ending with `.unit`) for processing by the Knit tools. As described previously, unit files are separate from the files that contain your program's code.

If you look at the `hello.unit` file yourself, you will see that Knit supports C++-style comments in unit files. A comment either starts with "`//`" and runs to the end of the line, or starts with "`/*`" and ends with "`*/`". (A comment that begins with "`/*#`" and ends with "`#*/`" is a *doc comment*, as described in Section 3.2.)

```
// ‘Hello’ is the (atomic) unit that describes our program.  It imports some
// I/O services (function ‘printf’) and exports ‘main’.
```

The comment says that `Hello` is the unit definition for the entire “Hello, World!” program. (The unit is “atomic” because it is implemented by a set of files, rather than by a set of other units.) Let us take a closer look at the parts of this definition.

4.1.3 Imports and Exports

A unit has a set of *imports* and a set of *exports*. (Both the **imports** and **exports** parts of a unit definition are required, even if one or both are empty.)

- Imports are the names of functions and variables that must be supplied to the unit, in order for the unit to be used. In our simple example, the imports of the `Hello` unit will come from the runtime environment, i.e., the C runtime library. In future examples, however, we will see that most unit imports are resolved by connecting one unit’s imports with another unit’s exports.
- Exports are the names of the functions and variables that are defined by the unit and provided for use by other units. The exports of a program’s “topmost” or “outermost” unit are available to the runtime environment. In the current example, the `Hello` unit is the topmost (and only) program unit.

Because a unit may import many items and export many items, Knit allows you divide your imports and exports into groups of related items, called *bundles*. An **imports** or **exports** specification contains a list of bundles definitions, which looks something like a list of variable declarations:

```
imports [ bundle-name1: bundle-type1 [, bundle-name2: bundle-type2, ...] ];
exports [ bundle-name1: bundle-type1 [, bundle-name2: bundle-type2, ...] ];
```

The name of a bundle (a symbol) appears to the left of the colon, and the type of the bundle appears to the right. Multiple bundle declarations are separated by commas. In our `Hello` unit, the type of each bundle is given as a list of names enclosed in braces, indicating the names of the objects being imported or exported from the unit:

```
  imports [ io: {printf} ];
  exports [ main: {main} ];
```

`Hello` has a single import bundle named `io`: this bundle has a single member `printf`. (If there were multiple members in the bundle, one would put commas between the member names.) Similarly, the exported bundle is `main` and contains a single member, also called `main`. It is not a problem to reuse names in this way, because the names of bundles and the names of bundle members are kept in separate namespaces. This is similar to the handling of variable names and `struct` member names in C: the names are in separate namespaces, and so will never conflict.

The names of bundles may be used in subsequent parts of the unit description. For instance, we may use them in the **depends** part of the unit, which we describe next.

4.1.4 Dependencies

The **depends** part of a unit definition states the dependency relations that exist between the imports and exports of a unit. Knit needs this information in order to schedule the initialization and finalization of units. For instance, the `Hello` unit says the following:

```
depends { main needs io; };
```

The declaration "`main needs io`" says that the functions in the `main` bundle make use of the functions from the `io` bundle. Thus, any initializers that are associated with the `io` bundle must be run before any functions from the `main` bundle can be called.

While we do not use Knit's scheduling features in our current example, it is always a good idea to describe the dependencies that exist in a unit: this information may be needed in other programs that incorporate your units. For this reason, Knit *requires* that all atomic unit definitions contain a **depends** clause.

We will describe dependencies in greater detail when we deal with initialization and finalization later in this tutorial (Section 4.3). Now, however, we describe the final part of our `Hello` unit.

4.1.5 Files

The implementation of the `Hello` unit comes from the file `hello.c`, as described in the unit definition:

```
files { "hello.c" };
```

If our program were more complex, we could list more than one source file in the **files** part of our unit definition. (Multiple file names would be separated by commas.) Knit needs the names of the implementation files in order to produce the `knit_generated.mk` file, which will contain the `make` rules for compiling our unit.

So, to sum up, our `Hello` unit definition says that the code in `hello.c` implements a function called `main`. The `main` function calls `printf` (as stated in the **depends** clause), and `printf` is imported from outside the unit. Now that we understand what the definition says, we are ready to process the unit file with Knit in order to create the "Hello, World!" program.

4.1.6 Compiling the `hello` Program

Assuming that you followed the Knit configuration instructions in Section 2.3, you should have an `examples/hello` directory in your Knit build tree. That directory should contain a `GNUmakefile` that you can use to run Knit and compile the `hello` program. (Note that the `GNUmakefile` reads a separate file called `GNUmakerules`, which is located in the Knit *source* tree, not in your build tree.)

Go into the `examples/hello` directory of your Knit build tree and type "`make`" or "`make all`". Assuming that you are starting from a blank slate (i.e., the program has not been built already), the build will start by running a command like this:

```
../../bin/knit \
  UNIT_PATH=.../examples/hello ... \
  hello.unit Hello
```

The command line syntax of the `knit` compiler is described in Section 3.1.2. In brief, the above command tells `knit` to process the `hello.unit` file and create the set of output files that are needed in order to compile an instance of the `Hello` unit. When `knit` finishes, it will have produced three files:

`knit_generated.mk`	The `make` rules that describe how to compile an instance of `Hello`. If you look at this file, you will see that the compiled unit code is put into an archive (‘`.a`’) file, and that the archive file is listed in the `KNIT_LIBS` macro.
`knit_inits.c`	A C file that contains Knit-generated initialization and finalization functions for our program. Since we do not use initializers or finalizers in this example, the generated functions are pretty uninteresting.
`rename_Hello`	A file that describes how certain symbols in our unit's object file will be renamed. This file is referenced from the rules in `knit_generated.mk`.

After `knit` has completed, the `make` process will continue: `make` will read the newly created `knit_generated.mk` and proceed to compile the `knit_inits.c` file.

```
gcc -c knit_inits.c -g -O2 -Wall -Wshadow
```

Note that `knit_inits.c` is *not* part of the `Hello` unit. Rather, it is part of the "runtime environment" for the unit.

Next, `make` will produce the archive file that contains the unit code:

```
gcc -g -O2 ... -o Hello_hello.c.raw.o -c .../examples/hello/hello.c
cp Hello_hello.c.raw.o Hello_hello.c.o
../../bin/rename_dot_o_files 'Hello_' rename_Hello Hello_hello.c.o
ar csq foo0.a Hello_hello.c.o
```

The actual `make` issues a few additional commands, which we have omitted to improve readability. The idea is to compile each source file, run the `rename_dot_o_files` tool to transform the object files as needed, and then combine all of the objects into an archive.

Finally, `make` will link the compiled unit, the `knit_inits.o` file, and the standard C runtime library to create the `hello` program:

```
gcc -o hello --begin-group knit_inits.o foo0.a --end-group
```

There are two important things that you might notice about this final command.

First, remember that the `Hello` unit is defined to import a `printf` function and export a `main` function. In our case, since `Hello` is our top-level unit, these symbols will be imported from and exported to the "environment" of our unit code, i.e., the standard C library and any other object files that we link into our program. If we had not imported `printf`, then *our program would not link*. Symbols from the environment are not implicitly imported into a unit: rather, they must be explicitly imported. Similarly, if we had not exported `main`, *our program would not link*, because the C library would not have access to the `main` function defined in our unit.

Second, the link command line lists the objects and libraries for our program between `--begin-group` and `--end-group`. As previously described in Section 3.1.3, this simply helps to avoid problems that the linker might have in resolving symbols, and eliminates the need for us to carefully order the files on the link command line. There is no "deep magic" here; it is simply convenient practice.

Before moving on to the next example, you should check that everything actually worked:

```
[10] examples/hello> ./hello
Hello, world!
```

4.2 Using Multiple Units: The `msg` Example

Our second example is similar to the "Hello, World!" program that we just built, except that our new program is built by combining two separate units. We will call our new program "`msg`". Let us start by looking at the source code for the main program, which is in the file `examples/msg/main.c` in the Knit source tree. The interesting part of the file is this:

```
const char *message();

int main(int argc, char** argv)
{
        printf("%s", message());
        return 0;
}
```

This is of course nearly identical to the `main` function of the "Hello, World!" program, except that now, the string to be printed is returned by an external function called `message`. Since this is a Knit tutorial, we of course want to get the `message` function from another unit. That other unit will export the function, and our main program unit will import it.

4.2.1 Bundletypes

It is not a problem to define multiple units: simply put multiple **unit** definitions in your unit file. But if the units are designed to be linked together, how can we best ensure that all of the various import and export bundles have the appropriate types? In the `hello` example (Section 4.1.3), you learned that import and export bundles can be written like this:

```
import [ bundle-name: { member1, member2, ... } ];
```

In other words, you list all of the members of the bundle between braces. This style is tedious, however, if you want to use the same kind of bundle more than one place — which is usually the case, after all, because most bundles are exported from one unit and imported into another! So, to help you avoid errors and verbosity in your unit definitions, Knit allows you to define bundle types by name. For example, in the unit file for our current program (`examples/msg/msg.unit`), you will see this:

```
// Define our ``bundletypes.''  A bundle is like an ``interface'': a set of
// functions that describe the imports or exports of a unit.
//
// Our bundletypes are exceedingly simple, since each contains only a single
// member.  In general, a bundletype contains several members and describes a
// group of related functions.
//
bundletype IO_T         = { printf }
bundletype Msg_T        = { message }
bundletype Main_T       = { main }
```

These definitions define three bundletypes in the obvious way. We can now use the bundletypes to define the unit that will contain our main program code, i.e., the code in `main.c`:

```
// ‘Main’ is the unit that encapsulates our ‘main’ function.  If you look at
// the code in ‘main.c’, you will see that ‘main’ calls ‘printf’ and ‘message’.
// We import those functions in two bundles (‘io’ and ‘msg’).
//
unit Main = {
    imports [ io: IO_T,
              msg: Msg_T ];
    exports [ main: Main_T ];
    depends { main needs (io+msg); };
    files { "main.c" };
}
```

This unit definition should look familiar, since it is very much like the definition of the `Hello` unit from the previous example (in Section 4.1.2). The three main differences are that:

1. `Main` import two bundles instead of one.
2. `Main` uses named bundle types instead of listing the bundle members explicitly.
3. `Main` uses a set-like syntax to say that its exported `main` bundle needs both of the imported bundles.

4.2.2 Renaming

Now we turn our attention to the second unit in our example: namely, the unit that will provide the definition of the `message` function. This unit will be implemented by the code in the `examples/msg/messages.c` file. If you look at that file, you will see that it defines three functions, each of which returns a string:

```
const char *not_worth_knowing()        { return "..."; }
const char *rarely_fits()              { return "..."; }
const char *change_the_spec()          { return "..."; }
```

Unfortunately, although all of these functions have the same C type as the `message` function we need, none of the functions at hand are actually called `message`! This kind of problem is often encountered by programmers who need to combine code from different sources. The usual C solution is to write miniature wrapper functions or to use C preprocessor magic to establish the wanted connections between functions. Unfortunately, these solutions are often tedious and break down at large scale.

With Knit, however, we can do better. Without changing the C code, we can define a unit that exports *every one* of these functions as a different instance of a `message` function. We will later decide which of these instances will be “the” `message` function that is imported into our `Main` unit.

To export each of our three functions as an instance of `message`, we define a unit that exports three bundles, each of type `Msg_T`. Then, we use **rename** declarations to say which C functions correspond to which bundle members, like so:

```
unit Messages = {
    imports [];
    exports [ msg_1: Msg_T,
              msg_2: Msg_T,
              msg_3: Msg_T ];
    depends { exports needs imports; };
    files { "messages.c" };
    rename {
        msg_1.message to not_worth_knowing;
        msg_2.message to rarely_fits;
        msg_3.message to change_the_spec;
    };
}
```

All three exported bundles are of type `Msg_T`, so each one exports a `message` function. By default, Knit automatically associates each imported or exported bundle member with a C function of the same name. But since that default rule cannot work here, we must make explicit pairings. Each **rename** declaration in our unit has the form:

```
rename bundle-name.member to c-function;
```

These declarations have the “obvious” effects. The `not_worth_knowing` function is exported as the function referenced by the `message` member of the `msg_1` bundle. The other two functions are referenced via the `message` members of `msg_2` and `msg_3`. Later in this tutorial you will learn how to rename several functions at once, but for now, we proceed with the current example.

4.2.3 Compound Units

At this point we have two units, `Main` and `Messages`. Each of these units is *atomic*, meaning that each is implemented by one or more source files. What remains in this example is to connect our units together, via a *compound* unit, to form a single unit that we can use as the top-level for our program.

A compound unit is very similar the units we have seen so far: for instance, a compound unit has **imports** and **exports**. However, instead of a **files** section, a compound unit has a **link** section. The **link** section states the units that make up the compound unit, and further, defines how these “internal” units are connected to each other and to the imports and exports of the compound unit itself.

For our current program, we need a compound unit that connects an instance of our `Main` unit with an instance of our `Messages` unit. If you look in the `msg.unit` file, you will find the following definition of the compound unit we need:

```
unit Msg = {
    imports [ io: IO_T ];
    exports [ main: Main_T ];
    link {
        [main]                  <- Main         <- [io, msg_1];
```

```
          [msg_1, msg_2, msg_3]   <- Messages     <- [];
     };
}
```

The **imports** and **exports** should look familiar. As explained in Section 4.1.6, since `Msg` is going to be the top-level unit for our program, we must explicitly import the services that we need from the environment (in this case, the functions listed in the `IO_T` bundletype) and explicitly export the functions that the runtime needs to invoke (i.e., our `main` function).

The **link** section of a compound unit describes how the unit is implemented in terms of a network of other units. Each statement in the link section above is of the form:

```
[ export₁, export₂, ... ] <- unit <- [ import₁, import₂, ... ]
```

Each line causes an instance of the named unit to be created. Let us take a closer look at the first line in the **link** part of our `Msg` unit definition. That line says that a `Main` unit instance will be created as part of the (compound) `Msg` unit. At the start of that line, "`[main]`" is a list of symbols: these give names to the bundles that are exported by our `Main` unit. Bundles are named in the order they are listed in the **exports** list of the unit being instantiated. (Of course, our `Main` unit has only one exported bundle.) At the end of the line, the list "`[io, msg_1]`" gives names to the imported bundles. Again, the bundles are named in the order they are listed in the `Main` unit definition.

Connections between units are indicated when the same name is used at two or more places in the compound unit. Looking again at the first line within the **link** part of `Msg`, we see that bundle being exported from `Main` has the same name as the bundle being exported from the `Msg` unit itself. This indicates that the export of `Main` is connected to the export of `Msg`: in other words, `Msg` exports the functions from its internal `Main` unit. Similarly, the `io` bundle that is imported to `Main` is connected to the `io` bundle that is imported by `Msg`.

So finally we see which of the functions in our `Messages` unit becomes "the" `message` function to be called in our program. The second link of our **link** specification gives names to the three bundles that will be exported from an instance of our `Messages` unit. One of these bundles (`msg_1`) is specified as an import to our `Main` bundle. Thus, the "wiring" in our compound unit tells us that the function that implements `msg_1.message` will be the one to actually be called in our program. (The bundles `msg_2` and `msg_3` are not connected to any other units, nor are they exported from the compound unit. This is not a problem — the bundles are simply unused. Also note that the `Messages` unit requires no imports, and so its import list is empty.)

Now it should be clear that you can easily change the "wiring" of the program, *without changing the C source code*. If you simply replace the `msg_1` import to `Main` with either `msg_2` or `msg_3`, you effectively change the message string that will be output by the complete program. This kind of flexibility is critical when building programs from components: in Knit, the linking *specifications* are separate from the component *implementations*.

4.2.4 Compiling the `msg` Program

Finally, you are ready to compile the `msg` program. Go to the `examples/msg` directory of our Knit build tree and type "`make`" or "`make all`". The `make` process will go through the steps described previously in Section 4.1.6, and the result will be program called `msg`. Run it:

```
[11] examples/msg> ./msg
A language that doesn't affect the way you think about programming is not
worth knowing.
```

If you re-examine the `msg.unit` file, you should be able to see why the program prints the message shown above, and not some other message. At this point, you might want to experiment by editing `msg.unit` to change the message output by your program. After a change to the unit file, a simple "`make`" should be all that is required to re-Knit and recompile your program. (Do not change the name of the `Msg` unit, however! If you change that name, you will have to edit the `GNUmakefile` in your build directory to match.)

4.3 Knitting Tricks: The `calc` Example

Now that you have mastered Knit basics, it is time to see how Knit can help in the development of a nontrivial C program. In this example, we will use Knit to define, build, and analyze a four-function expression evaluator — in other words, a calculator. The basic program will read expressions from the user, evaluate them, and print out the results:

```
[12] examples/calc> ./calc
1+2
read    : 1 + 2
eval    : 3
```

To make things a little more interesting, we will Knit together a special version of the program that monitors calls to `malloc` and `free` for two different datatypes in the program. The enhanced program will report its allocation statistics for each input expression, like so:

```
[13] examples/calc> ./calc
1+2
read    : 1 + 2
read    : (allocs/frees)  4/ 4 tokens,  3/ 0 exprs
eval    : 3
eval    : (allocs/frees)  0/ 0 tokens,  3/ 2 exprs
cleanup : (allocs/frees)  0/ 0 tokens,  0/ 4 exprs
total   : (allocs/frees)  4/ 4 tokens,  6/ 6 exprs
```

As shown in the transcript, in the "read" phase of the program, four `token` objects were allocated, four `tokens` were freed, three `exprs` were allocated, and zero `exprs` were freed. Similar statistics were reported for the "eval" and "cleanup" phases. Finally, the "total" line shows the sums of the counts from the three phases. In the example shown, for both `tokens` and `exprs`, the number of allocs is equal to the number of frees. This is good evidence that there were no memory leaks.[1]

The C code for our calculator — approximately 1000 lines — is located in the `examples/calc` directory of the Knit source tree. The `calc.unit` file organizes the calculator as a small number of atomic units — one for each major component — and links them together using compound units. If you have worked through the previous examples in this chapter, you should already understand most of the contents of the `calc.unit` file. Therefore, in the sections below, we describe only the Knit language features that were not used in the `hello` or `msg` programs.

[1]The evidence is not conclusive, however, because our program counts the number of calls to the allocation and free functions, but not operations on individual objects. A more sophisticated program would track individual objects, but that is not the point of our example.

4.3.1 Initializers and Finalizers

The first new Knit language feature in our example is the use of *initializers* and *finalizers*. In Knit, a component can specify one or more functions that must be called to initialize the component — more precisely, to initialize one or more of the component's exports. Similarly, a finalizer is a function that must be called in order to shut down some of the component's exports.

Why does Knit treat initializers and finalizers in a special way? Why not simply list initializers and finalizers among a unit's imports and exports? It is for the same reasons that languages like C++ have special notions of constructors and destructors:

- *Simplicity.* By handling initializers and finalizers specially, Knit can ensure that they are run — instead of leaving the task in the hands of each Knit user.
- *Correctness.* Moreover, Knit can ensure that initializers and finalizers are run in a correct order. For instance, if one unit's initializer invokes functions that are imported from a second unit, then Knit will ensure that the second unit is initialized before the first unit.
- *Modularity.* Finally, Knit's handling of initializers and finalizers helps to make components more modular. Two units providing the same exported bundletypes may have very different initialization and finalization requirements inside. By hiding this difference, Knit makes it possible for a programmer to use one unit in place of the other: there is no need for the programmer to write or change any initialization code by hand. By handling initializers and finalizers automatically, Knit provides better separation between a unit's interface and its implementation.

The syntax for specifying initializers and finalizers is illustrated by the `Input` unit in our `calc` example:

```
unit Input = {
    imports [ alloc     : Alloc_T,
              io        : IO_T ];
    exports [ input     : Input_T ];

    initializer init_input for exports;
    finalizer fini_input for exports;
    depends {
        // { init_input } is syntax for ‘‘the set containing ‘init_input’.’’
        { init_input }  needs io;
        { fini_input }  needs io;
        exports         needs imports;
        //
        // As described previously, if we wished, we could replace the above
        // three lines with a single (overgeneral) statement that all of our
        // exports, initializers, and finalizers depend on all of our imports:
        //
        //      (exports + inits + finis) needs imports;
    };
    files { "input.c" };
}
```

In the above definition, the C function `init_input` is specified to be an initializer for all of the unit's exports (as indicated by the keyword `exports`). Similarly, the C function `fini_input` is the finalizer for all of the exports. In general, one can provide a specific set of bundles when defining an initializer or finalizer, but it is usually sufficient to say simply that the function is an initializer or finalizer for all exports. Moreover, it is often a good idea to overgeneralize in this way. If you later tweak the C code and add a new unit export, for example, you do not have to remember to specify that your initializer or finalizer also applies to the new export. Finally, note that initializers and finalizers do not generally need to be exported: Knit invokes them specially. (The only reason to export an initializer or finalizer would be if you want Knit to invoke the function automatically *and* you want to explicitly invoke it yourself. This would be rather odd.)

So how are initializers and finalizers used? When the `knit` compiler is run, it creates a file called `knit_inits.c` that contains two function definitions. The first function, `knit_init`, contains a list of calls to the initialization functions for the unit instances within your program.[2] The second function, `knit_fini`, contains calls to the finalizers in your program.

The `knit_init` function must be called before your program proper, i.e., before any of the exports from your program's top level unit are called. In the current example, this is accomplished with some "runtime magic" in the `init.c` file. Pay special attention: the `main` function of the calculator program does *not* invoke the initializers! Instead, the Knit runtime support in `init.c` ensures that the `knit_init` function is run before `main` is called. Similarly, the code in `init.c` ensures that `knit_fini` will be called after the top-level exports (in this example, the `main` function) will no longer be called (i.e., after `main` has returned, or `exit` has been called).

4.3.2 More About Dependencies

The order of the calls in the Knit-generated `knit_init` and `knit_fini` functions are based on the dependency information found in your unit definitions. For instance, if one initializer needs to call functions that are imported from a second unit, then the second unit must be initialized before the first. Accurate (or, at least, conservative) dependency information in all units is a *must* in order for Knit to find correct initialization and finalization schedules. This is why dependency information is required even for atomic units that do not themselves have initializers and finalizers, as was previously described in Section 4.1.4.

If we look again at the **depends** section of our `Input` unit, you will notice some new syntax for describing dependencies:

```
depends {
    // { init_input } is syntax for ‘‘the set containing ‘init_input’.’’
    { init_input }  needs io;
    { fini_input }  needs io;
    exports         needs imports;
    //
    // As described previously, if we wished, we could replace the above
    // three lines with a single (overgeneral) statement that all of our
    // exports, initializers, and finalizers depend on all of our imports:
    //
    //      (exports + inits + finis) needs imports;
};
```

[2]More precisely, `knit_init` contains calls to the initializers for the unit instances that make up the top-level unit that was specified on the `knit` compiler command line.

The first piece of new syntax is for “object sets” as illustrated in the first two statements. To specify that the `init_input` and `fini_input` functions each call functions from the imported `io` bundle, we create object sets by putting the function names in braces as shown. Note that we could have put both functions in a single set. Also note that we must use the object set syntax here, because our initializer and finalizer functions are not part of any named (imported or exported) bundle.

The second piece of new syntax is illustrated by the third statement. Instead of naming specific bundles, a dependency statement can refer to certain predefined groups of bundles:

`exports`	All members of the unit’s export bundles.
`imports`	All members of the unit’s import bundles.
`inits`	The set of all of the unit’s initializers.
`finis`	The set of all of the unit’s finalizers.

Further, Knit allows the unit writer to combine object sets using “`+`” for set union and “`-`” for set difference, as shown in the comments above. As the comments describe, we could replace all of the dependency statements in the `Input` unit with the single statement:

```
  (exports + inits + finis) needs imports;
```

which conservatively approximates (overgeneralizes) all of the actual dependencies in the unit. When writing your own units, it is often good to start with the above statement — but, be careful! If dependency information is *too* conservative, Knit may find an *initialization cycle*: a cycle of units in which each unit requires that the previous unit to be initialized before initializing itself. This can happen if there is in fact a true dependency cycle, or if your units’ dependency specifications are too general (so that they introduce false dependency cycles). In the latter case, you will need to make your dependency specifications more accurate, so that Knit can find workable initialization and finalization sequences. Fortunately, both true and false dependency cycles are rare in most programs.

Before moving on to further discussion of renaming, it would be useful for you to read through the definition of the `Alloc` unit in our calculator unit file. The (rather long) comments in the **depends** section in particular clarify the relationship between dependencies and initializers. (In case you do not have the file handy right now, the lesson is this: it is extremely unusual for a bundle to depend on its initializer.)

4.3.3 More About Renaming

If you just read through the `Alloc` unit definition as suggested above, you may have noticed some new syntax for renaming:

```
    rename {
        // We need to associate ‘counted_alloc.malloc’ with the C function
        // ‘counted_malloc’, and likewise for ‘counted_alloc.free’.  To make
        // these associations, we could use two separate renaming declarations:
        //
        //      counted_alloc.malloc    to counted_malloc;
        //      counted_alloc.free      to counted_free;
        //
        // But we can do the same job by saying that the C function names are
        // derived by adding a prefix to the names of the bundle members:
        //
        counted_alloc   with prefix counted_;
    };
```

When we previously discussed renaming in Section 4.2.2, we learned how to make associations one-by-one. To make certain common cases easier, however, Knit provides special syntax for renaming when the names of C functions can be manufactured by adding a prefix or suffix to the names of the members of a bundle. The syntax of these special cases is:

```
rename {
  bundle-name with prefix identifier;
  bundle-name with suffix identifier;
};
```

Of course, this convenient syntax is useful only for prefix or suffix transformations. You cannot apply both a prefix and a suffix. For situations requiring more that a simple prefix or suffix addition, you must use Knit's one-by-one syntax.

4.3.4 Wrappers and Transparent Interposition

For the allocation-monitored version of the `calc` program, we want to count the numbers of `expr` and `token` objects that are dynamically allocated and freed. Further, we want to count these events separately for each type. Let us consider `exprs` first. If you look at the code in the `expr.c` file, you will find the `alloc_expr` function, which handles all dynamic allocations of `exprs`:

```
static expr
alloc_expr(void)
{
        return ((expr) malloc(sizeof(expr_struct)));
}
```

There is an analogous `free_expr` function for handling dynamic releases; `free_expr` invokes the standard `free` function to actually release the memory for a given `expr` object.

Counting the number of dynamic `expr` allocations and frees, therefore, amounts to counting the number of times that `alloc_expr` calls `malloc` and the number of times that `free_expr` calls `free`.[3] More precisely, we must count the number of times that `malloc` returns a non-null result and the number of times that `free` is called with a non-null argument. To do this, we need to *interpose* on or *wrap* calls to these functions, so that we can insert our instrumentation.

[3] If you look at the code, you will see that it is not correct to count the number of calls to `free_expr` itself, because `free_expr` does not always call `free`.

Given this scenario, most C programmers would either (1) edit the `alloc_expr` and `free_expr` functions to insert the needed instrumentation, or (2) define C macros to "magically" replace the calls to `malloc` and `free` with calls to other functions. Each of these solutions has its problems, however.

The first technique requires the programmer to edit the code. Likely, the programmer will complicate the code with `#ifdefs` so that the instrumentation can be conditionally incorporated into the program. While doing this once or twice might not be a problem, doing it many times turns the code into an `#ifdef` jungle!

The second approach — instrumentation via macro magic — has a similar but different problem. It is easy enough to define `malloc` and `free` as macros that call other functions, say, `counted_malloc` and `counted_free`. To use these macros, the programmer would probably need to change only a few `#include` lines in the source, so while source changes are still required, they are minimal. A new problem arises, however, when the programmer remembers that we want to monitor *both* `expr` and `token` allocation, and that we want *separate* counts for each type! Now we cannot use our simple macros to insert instrumentation into both `expr.c` and `token.c`, because we need slightly different instrumentation for each file.

A possible solution would be to make more complicated macros, e.g., macros that expand differently based on other macros. But you do not want to do that. You want to use Knit, which can solve your problems in an elegant and principled way.

In the calculator program at hand, the code in `expr.c` is encapsulated by the `Expr` unit, which is defined in the `calc.unit` file. That unit says that the functions `malloc` and `free` are imported into the unit as elements of a bundle called `alloc`, as shown in this excerpt:

```
bundletype Alloc_T      = { free, malloc }

unit Expr = {
    imports [ alloc             : Alloc_T,
```

You, Knit user, decide where these functions come from. They do not have to come from the standard C library: they can come from *any* unit that exports a bundle of type `Alloc_T`. Moreover, the choice is *transparent* to the code in the `Expr` unit: the code in the `Expr` does know or care where the allocation functions come from. Thus, we can transparently replace the standard allocation functions with monitored versions of those functions, if we have a unit that implements the counted versions we want.

Fortunately, we have such a unit: `Alloc`. The `Alloc` unit provides versions of the allocation functions that count the number of times that they allocate or free memory. Functions to get and reset the allocation counts are provided in a separate exported bundle, as shown in the excerpt below:

```
unit Alloc = {
    imports [ alloc             : Alloc_T ];
    exports [ counted_alloc     : Alloc_T,
              counts            : AllocCounts_T ];
```

Note that `Alloc` both imports *and* exports bundles of type `Alloc_T`. This is a common Knit idiom for a unit that *wraps* another unit: the "wrapper" unit modifies or otherwise interposes on access to the inner "wrapped" unit. In this case, the `Alloc` unit imports definitions of `malloc` and `free`, and then exports its own versions of these functions. (A **rename** declaration, which we discussed previously in Section 4.3.3, is required in order to export the unit's own definitions as bundle elements called `malloc` and `free`.)

4.3.5 Multiple Instantiation

As described in the previous section, the code in `expr.c` is encapsulated by the `Expr` unit, and the `Expr` unit imports the functions `malloc` and `free` from the outside, i.e., some other unit. Now, notice that the code in `token.c` is encapsulated in the `Token` unit, and that like `Expr`, the `Token` unit imports the allocation functions from another unit:

```
unit Token = {
    imports [ alloc             : Alloc_T,
```

The key insight here is that, although both `Expr` and `Token` import allocation functions, *they do not have to import these functions from the same unit*. Instead, every unit instance can import these functions from a separate unit instance. The `Expr` unit can get allocation from one unit instance, and the `Token` unit can get different allocation functions from a different unit instance. In this way, we can effectively instrument our `Expr` and `Token` units separately, without changing the C source code of either unit.

The obvious solution, then, is for us to put *two* separate instances of our `Alloc` unit into our final calculator program: one to track the behavior of expression objects and the other to track the behavior of tokens. This approach gives us separate counts for `expr` and `token` objects, which is what we want — but not everything we want. Remember that in addition to tracking allocation for the two type separately, we also want to track allocation behavior both for each "phase" of the interpreter, and for each input expression as a whole (i.e., the "total" counts in transcript below).

```
[14] examples/calc> ./calc
1+2
read    : 1 + 2
read    : (allocs/frees)  4/ 4 tokens,  3/ 0 exprs
eval    : 3
eval    : (allocs/frees)  0/ 0 tokens,  3/ 2 exprs
cleanup : (allocs/frees)  0/ 0 tokens,  0/ 4 exprs
total   : (allocs/frees)  4/ 4 tokens,  6/ 6 exprs
```

In other words, what we really need are two sets of numbers for each type: one set of numbers that we clear between phases of the interpreter, and a second set that we clear only between expressions. The `Alloc` unit in our unit file defines a unit that exports allocation functions and one set of allocation counters. Cleverly, we can use this unit to create a unit exports allocation functions and *two* sets of counters, simply by composing two instances of `Alloc` as shown below:

```
unit Alloc_2 = {
    imports [ alloc             : Alloc_T ];
    exports [ counted_alloc     : Alloc_T,
              counts_1          : AllocCounts_T,
              counts_2          : AllocCounts_T ];

    link {
        // [export, export, export, ...]
        //   <- Unit
        //   <- [import, import, import, ...];

        // The exported ‘counted_alloc’ and ‘counts_1’ bundles, from one
        // instance of an ‘Alloc’ unit.
        [counted_alloc, counts_1]
            <- Alloc
            <- [counted_alloc_internal];
        // The internal ‘counted_alloc_internal’ bundle and the exported
        // ‘counts_2’ bundle, from a separate instance of our ‘Alloc’ unit.
        [counted_alloc_internal, counts_2]
            <- Alloc
            <- [alloc];
    };
}
```

`Alloc_2` defines a unit like `Alloc`, but with two separate sets of counters. Each counter set is accessed by a bundle of functions: the first set by elements of `counts_1`, and the second set by the elements of `counts_2`. The `Alloc_2` unit is implemented as a compound unit (Section 4.2.3) that connects two instances of `Alloc` in the "obvious" way. The exported allocation functions from one instance of `Alloc` are given as imports to the second instance of `Alloc`. The exported allocation functions from the second instance are then exported from `Alloc_2` itself. The bundles for accessing the counts are also exported from `Alloc_2`, thus creating the two-count-set unit that we need for our calculator program.

When a unit is instantiated more than once, *each instance of the unit is independent*. This is true whether the unit is instantiated multiple times within a single containing unit (as shown above) or multiple times as parts of different containing units. In either case, each instance of a unit has its own imports, its own exports, and therefore, its own copy of its code and data (e.g., static variables declared in the unit's C files). In terms of "objects," you might think of each unit instance as a separate object instance, with its own relationships to other units. In terms of linking, you might think in terms of the object code being linked multiple times into the final program, although tailored for each individual copy.

A careful reading of the `Alloc_2` unit definition provides additional detail about the behavior of the two counter sets. The two sets of counters are *correlated*: for each call to one of the exported allocation functions, *two* counter instances will be incremented. But, by looking closely at the wiring within `Alloc_2`, one can see that the two sets of counters are *independent*: i.e., that neither depends on the values of the other, and that if you reset one of the counter sets, the other counter set will be unaffected. You can see this because neither `Alloc` unit imports the other's `counts` bundle, and therefore, neither unit could possibly invoke the `get_alloc_counts` or `reset_alloc_counts` functions on the other.

4.3.6 Summary: The `Calc` and `Calc_Counted` Units

So, finally, we have what we need in order to build an instrumented version of the calculator program! The `calc.unit` file contains top-level units for both the "plain" and instrumented versions of the program: these units are called `Calc` and `Calc_Counted`, respectively. Let us briefly summarize the important parts of `Calc_Counted`:

- `Calc_Counted` is the compound unit that instantiates and connects all of the components within the instrumented calculator program. This is the unit that we will eventually name of the command line to the `knit` compiler.

- `Calc_Counted` instantiates two copies of the `Alloc_2` unit: one for tracked allocation of `exprs` and another for tracked allocation of `tokens`. Each instance of `Alloc_2` contains two instances of `Alloc`; thus, there are four separate instances of `Alloc` in the final program.
- Each instance of `Alloc_2` imports the allocation functions that are imported by `Calc_Counted` itself. Ultimately, the implementations of these functions comes from the runtime environment outside Knit, i.e., the standard C library.
- The (exported) instrumentation bundles from the two `Alloc_2` instances are connected (imported) to an instance of `Repl_Counted`, which is the main "read-eval-print-loop" for the instrumented program. This gives the main loop the ability to access and clear the allocation statistics as it needs.

The complete definitions of the `Calc` and `Calc_Counted` units are located at the end of the `calc.unit` file. By now, everything in these unit definitions should be clear to you — except for the **flatten** directives, which we will describe below in Section 4.3.8.

4.3.7 Compiling the `calc` Program

A simple "`make`" or "`make all`" in the `examples/calc` directory of your Knit build tree will run `knit` to produce the instrumented version of the calculator program. The `make` process will go through the steps described previously in Section 4.1.6, and the result will be program called `calc`. In the output from `make`, you may notice that `knit` prints out the schedule for the program's initializers and finalizers.

Run your newly compiled `calc` program:

```
[15] examples/calc> ./calc
1+2
read    : 1 + 2
read    : (allocs/frees)  4/ 4 tokens,  3/ 0 exprs
eval    : 3
eval    : (allocs/frees)  0/ 0 tokens,  3/ 2 exprs
cleanup : (allocs/frees)  0/ 0 tokens,  0/ 4 exprs
total   : (allocs/frees)  4/ 4 tokens,  6/ 6 exprs
```

As we described in the introduction to this example (Section 4.3), the statistics reported by `calc` give us confidence that the program is behaving correctly, without memory leaks. It turns out, however, that most nontrivial programs have bugs:

```
(1
read    : scanner error: unexpected end of input
read    : (allocs/frees)  3/ 3 tokens,  2/ 0 exprs
eval    : scanner error: unexpected end of input
eval    : (allocs/frees)  0/ 0 tokens,  1/ 0 exprs
cleanup : (allocs/frees)  0/ 0 tokens,  0/ 2 exprs
total   : (allocs/frees)  3/ 3 tokens,  3/ 2 exprs
```

Although the program correctly handled the erroneous input, it apparently leaked an `expr` object in the process: the "total" line indicates that `calc` allocated three `exprs` but freed only two. The author of the `calc` example found this bug quickly *because* Knit allowed him to easily insert instrumentation code into the program. This bug has been left in the C code in case you wish to examine it — see the function `parse_term` in the file `parse.c`.

At this point, you may want to experiment with the `calc` program. Here are some suggested exercises:

- Fix the above-described bug, then build and test a corrected version of the program.
- Edit the `GNUmakefile` so that Knit will build the uninstrumented version of the program. To do this, specify `Calc` as the top-level unit instead of `Calc_Counted`.

  Note that if you edit the `GNUmakefile`, you will need to "`make veryclean`" before building your new program. Otherwise, `make` may be confused by the files that were leftover from the previous run of `knit`.
- Use your knowledge of Knit wrapper units to test the `calc` program under simulated low-memory conditions. Start by writing special versions of `malloc` and `free` that simulate low-memory conditions. Your version of `malloc` should return a null pointer if the total number of allocated bytes would exceed some predefined limit, e.g., 128 bytes. (Of course, `free`'ing an object increases the budget for further allocation!) Then, write a Knit wrapper unit, and use it to test the instrumented `calc` program for bugs.
- Experiment with Knit's "flattening" optimization, described in the next section.

4.3.8 Optimizing the Code via "Flattening"

Flattening is an optimization technique in which the `knit` compiler "weaves" all of the C source files that make up a unit into a single ("flat") C file. The source code is manipulated to create the proper internal unit connections, of course, but is also manipulated so as to inline functions, remove dead (unused) functions, and hopefully improve the C compiler's ability to further optimize the code. The code transformations are heuristic, but work well for many cases.

Flattening is controlled by directives in your unit definitions. To say that the implementation of a unit should be flattened, insert the **flatten** directive into the unit's definition. (This directive is already specified in the `Calc` and `Calc_Counted` units of this example.) Flattening will be "recursively" applied to the entire body of the unit — including the bodies of units within compound units — except for units whose definitions include a **noflatten** directive. Through **flatten** and **noflatten** directives, you can specify which parts of your units are flattened and which are not. Finally, note that flattening directives are honored *only* when the optimization is enabled on the `knit` command line via the `-f` option. By default, flattening is not performed.

To apply flattening to the `calc` example, first "`make veryclean`" in order to remove any files previously created by Knit. Then, do:

```
make KNIT_FLAGS=-f
```

This will rebuild the program with flattening enabled. Because **flatten** is specified for the top-level program unit, the entire program will be flattened. The result is an optimized `calc` program — although you might not notice much speed improvement in this simple example!

Flattening can be "fine-tuned" by specifying an inlining budget. To specify a budget, add a value for `KNIT_BUDGET` to the list of flags for `knit`:

```
make veryclean
make KNIT_FLAGS='-f KNIT_BUDGET=1000'
```

As described previously in Section 3.1.2, *very roughly*, `KNIT_BUDGET` is the total number of static RISC instructions that should be spent on or saved by inlining. Positive values represent spending (i.e., increased code size) while negative values represent saving (reduced code size). To see a difference in the program size of `calc`, remember to strip the binary of debugging information. Since the `calc` program is relatively small, you should expect that any program size changes will also be small. Also note that the budget measures are *very* approximate: a larger budget may actually result in a smaller final program.

4.4 Other Knit Features

Although you have reached the (current) end of the tutorial, there are many features of Knit that we have not yet discussed. These things include:

- *Constraints*: the ability to specify constraints on the use of units, and have these conditions automatically checked by Knit.
- Defining bundletypes in terms of other bundletypes, and defining units as modifications of other units.
- *Inline units*, *anonymous units*, and matching bundle members by name in **link** specifications.
- *Globals*, *defaults*, *packages*, and *glue*.

For information about these language features, refer to the *Report on the Language Knit: A Component Definition and Linking Language*, which is contained in the `doc/report` directory of the Knit distribution. *Happy Knitting!*

Chapter 5

Debugging Knitted Programs

This chapter will be expanded in the future. For now, it only outlines the basics of what one must know in order to debug Knitted programs.

To debug a Knit-generated program, you must (1) compile the program for debugging, i.e., by giving the `-g` option to your C compiler, (2) run the Knit-generated program under a debugger such as `gdb`, and (3) understand how Knit has renamed ("mangled") the names of the objects — the variables and functions — that make up the program. The rest of this chapter describes how Knit mangles the names of objects.

Knit renames objects in different units to avoid name clashes. In particular, Knit prefixes every object name with a string that identifies the unit instance that contains the object. Knit assigns a unique name to every unit instance it creates as follows:

- **The topmost unit instance** — i.e., the instance of the unit named on the `knit` command line — has the same name as its unit definition.

- **A named unit instance within a compound unit** has a name of the form:

  *parent*_*bundle*

  bundle is the name of the first bundle exported by the unit instance. Within the **link** section of a compound unit definition, every unit instance exports at least one bundle, and all of the exported bundle names are unique. Therefore, *bundle* is a unique name within the compound unit.

  parent is the name of the "parent unit instance" — i.e., the name that Knit assigns to the instance of the containing compound unit. This name is, of course, determined by recursively applying these rules.

Every object that is not exported by the topmost unit instance is renamed by prepending the name of the unit instance in which the object is defined. The new name of the object is the name of this unit instance, followed by an underscore, followed by the object's original name. For example, consider the following unit definitions:

```
unit A = {
    imports [...];
    exports [...];
    link{ [v,w] <- PQ <- []; }
    link{ [x,y] <- PQ <- []; }
}

unit PQ = {
    imports [...];
    exports [ p: {p}, q: {q} ];
    ...
}
```

If `A` is named as the topmost unit, then the three unit instances in the Knit-generated code are assigned the names:

`A`	The topmost unit instance.
`A_v`	The first instance of `PQ` within the instance `A`.
`A_x`	The second instance of `PQ` within `A`.

The instances of `p` and `q` are assigned the names:

`A_v_p`	The object `p` in the unit instance `A_v`.
`A_v_q`	The object `q` in the unit instance `A_v`.
`A_x_p`	The object `p` in the unit instance `A_x`.
`A_x_q`	The object `q` in the unit instance `A_x`.

The four names above are the names that you would use within your debugger to set breakpoints or examine variable values.

Final notes:

- In fact, `gdb` will let you use the original name of a function if it is unambiguous, but always requires the “fully qualified” name for variables.
- You may find it faster to locate a symbol by executing a command like this:

  ```
  nm *.a | grep free_number
  ```

  instead of applying the above rules to determine the name of the unit instance.

Chapter 6

Knitting the OSKit

To use Knit with the OSKit, you should:

1. Obtain a matching version of the OSKit. (The Knit download page says which version of the OSKit to use with each version of Knit.)

2. Unpack, configure and prepare the OSKit:

   ```
   mkdir /tmp/src
   cd /tmp/src
   tar zxvf oskit<version>.tar.gz
   mkdir /tmp/obj
   cd /tmp/obj
   /tmp/src/oskit/configure --enable-knit
   make prepare
   ```

3. Try building some of the kernels:

   ```
   mkdir /tmp/test
   cd /tmp/test
   knit                                  \
     OSKITDIR=/tmp/src/oskit             \
     BUILDDIR=/tmp/obj                   \
     UNIT_PATH=/tmp/src/oskit/knit       \
     MAKEFILE=/tmp/src/oskit/knit/knit.mk \
     Delta.unit \
     Hello_ADR
   make -s
   ```

   This will build an OSKit kernel in the file `kernel`.

   The following units define OSKit kernels.

   ```
   Hello_ADR
   Hello_Delta
   Timer_Delta
   Timer1_COM_Delta
   ```

```
        Timer2_COM_Delta
        MemFS_COM_Delta
        Blkio_Delta
        DiskPart_Delta
        MemFS_Posix_Delta
        NetBSD_Posix_Delta
        PingReply_Delta
        Cat_Delta
```