MAKE IT!

Generating and Maintaining Makefiles Automatically

Sven Schönherr* Alexander Wolff*

Abstract

MAKE IT! automizes the process of writing makefiles for C and C++ projects. It supports the idea of literate programming, i.e. of keeping code and documentation in one file. From this file, a LaTeX file with the documentation and files with the program code are extracted. Then files are scanned for include statements in order to detect dependencies among them. These are written into so-called dependency files and subsequently used by the make process to ensure that products and documentation are up to date. The time overhead is small since a dependency file contains only the dependencies deduced from one file. Thus it can be reused until a change in this file occurs.

MAKE IT! simplifies software development and improves the resulting software in three ways. MAKE IT!

– frees the programmer's mind of the duty of writing and maintaining makefiles,

– considerably lowers the barrier to using literate programming, and

– offers full support for simultaneous work on different platforms with different compilers, which is a necessity for developping portable code.

1 Introduction

Most programming languages offer the programmer schemes to split up the source code of a project into smaller units. In C and C++, a project consists of *main implementation files*, and typically several *modules* and (pure) *header files*. A modul is split into two files, a header and an *implementation file*. The header file of a module by convention contains the declaration of a function or a data type. The implementation file of a modul then contains the implementation of the function (hence the name) or the definition of the data type declared in the header file. These data structures and functions are public in the sense that wherever they are needed, it is enough to include the corresponding header file to use them.

Generating *executables* from such a collection of source files is a two-step process. First, all implementation files are compiled, resulting in so-called *object files*. Then for each main implementation file all necessary object files are linked together.

On the one hand splitting up the source code has the advantage that it can be treated in smaller chunks by the programmer, the text editor and the compiler. Also, after an update only those files must be re-compiled which have changed. On the other hand more administration is

*Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin-Dahlem, Germany, {sven | awolff}@inf.fu-berlin.de

needed. Before the object files can be linked together, the programmer must make sure that they are all up to date, i.e. each must be more recent than the corresponding implementation file and all the header files that this implementation file includes. We say that an object file *depends* on these files.

The UNIX command `make` supports the programmer in this task. It has some built-in rules which use conventions about file name extensions (like `.h` for header, `.c` for implementation files) to generate object files, for example. `make` does not analyze the source code of files though, and thus only knows about generic dependencies, like that of an object on its implementation file. The programmer must collect all additional dependencies by hand in a so-called makefile. A makefile contains *rules*, and a rule typically consists of a target, the command that makes the target, and the targets on whose completion the rule's target depends. `make` reads these rules, recursively determines all dependencies, and then executes the commands that are necessary to update the *goal targets*, i.e. the targets the programmer specified.

Things became more complicated when we started to go into literate programming to improve both code and documentation of our projects [Knu84]. Now we first had to extract product files with the code from our *web* files before we could compile and link. We were rewarded with a more readable documentation. We use ANYWEB [Ket96], a front end to FUNNELWEB [Wil92], since ANYWEB uses more natural macro names (like `@file` and `@macro` instead of `@O` and `@$`, respectively) and is not fuzzy about the use of tabulators or blanks. We first translate an ANYWEB into a FUNNELWEB file (actually two different files), and then use FUNNELWEB to "tangle" product files and to "weave" the documentation.

After being fed up with writing makefiles for our growing programming projects we started trying to automize this error-prone process. In an intermediate stage of the development of MAKE IT!, we used hand-written lists of file names to declare the dependencies among the various parts of our projects, but we realized that this manual preprocessing could also be done by machine. So we wrote shell scripts which scan web and product files for include statements in order to detect dependencies. We had to consider the question how we could make the assembled information accessible to the make process – and in which chunks.

Hints in the documentation of GNU `make` [SM94] helped us to come up with the idea of *dependency files*. For each web file and for each implementation file we use such a dependency file to store all (non-generic) dependencies associated with the corresponding source. These dependency files are included by the *project makefile*. The project makefile is automatically copied to the project directory when the programmer types `make` there for the first time. It contains rules with generic dependencies and commands that have to be executed to create the targets. The project makefile does not change, only dependency files are added, deleted, or re-written when necessary.

We chose *environment variables* as an interface between MAKE IT! and its users. These variables are very versatile in that they can be set globally when the user logs in, but can be overwritten with project, platform or compiler dependent values. This makes it easy to change the platform or the compiler, especially in connection with modules [FO96].

Previous work. In [BV88] a makefile generator `mmf` for C programs is presented. `mmf` does not rely on the convention that external functions or data structures are only accessed by including the appropriate header file. `mmf` either parses C code (which must be syntactically correct) to find functions or data structures which are declared extern and thus imported from other files, or uses the UNIX command `lorder` to extract dependencies from existing object files. However, mmf is not as easy to use as MAKE IT!, does not support literate programming and does not split up the makefile in smaller units to allow fast updates. Running times were not available.

Structure. In the next section we present the main ideas of the design of MAKE IT!. In Section 3 we explain what happens when the user types make after setting a project under the control of MAKE IT!. In Section 4 and 5 we show how MAKE IT! keeps implementation and documentation of a project up to date. Finally, in Section 6, we present some experimental data of our own work with MAKE IT!.

In recent years, the esteem for computer experiments has increased even within theoretical computer science. We feel that new and better tools are needed to support good implementation of algorithms in order to narrow the gap between theory and practice. Our experience suggests that MAKE IT! is such a tool.

2 Design

make is a very powerful tool to keep all parts of a project up to date, but writing good makefiles is not an easy task; it is error-prone. Especially in the initial phase of a project, inter-file dependencies change and new files are often added. The need to maintain the makefile then distracts the programmer from the actual goal, namely developping good software. The file structure of a project might be considered more rigid than necessary, since one is tempted to trade in a slightly better design for not having to touch a working makefile – never change a winning team.

This is one of the points where MAKE IT! can help. It does not only free the programmer of the administrative duty to write and maintain makefiles, but also gives the programmer more freedom when setting up the file structure of a project. If it increases the oversight why not splitting up a directory into subdirectories – it only takes a call to make rescan and the reorganization is done. Correctly!

When designing MAKE IT!, we put emphasis on the following aspects.

Ease of use. After installing MAKE IT!, the user just has to type make in his project directory, and MAKE IT! will create all products and documentation – if there is at least one web or implementation file to process, and if it is where MAKE IT! expects it.

Apart from the usual targets make accepts, MAKE IT! offers special targets with which a whole group of files can be made or removed, e.g. make obj makes all object files, and make clean_ps removes all postscript documentation.

Customizability. MAKE IT! is driven by environment variables. Environment variables are used to tell MAKE IT! where to find libraries, web and implementation files, where to put product files, what to set their permissions to and how to recognize the type of a file by its extension. All these variables have default values which reflect common practice among programmers, e.g. the default header file extension is .h. These environment variables can be changed on four different levels: a change can be global and permanent, project or shell specific, or just be valid for a single call of make.

Platform independence. Whether you develop with ANYWEB, FUNNELWEB or (still) without literate programming, on a SUN Sparc or on a PC under Linux – MAKE IT! works. The compilers we used were CC-4.2 and the g++-2.7.2 family. There is one restriction, though. MAKE IT! heavily uses non-standard features offered by GNU make. GNU make is public domain [SM94].

Small overhead. MAKE IT! scans web and implementation files for include statements in order to detect dependencies. These are written into dependency files which are then included by the project makefile to ensure that products and documentation are up to date. Since a dependency file contains only the dependencies of one web or implementation file and can thus be re-used until a change in its source occurs, the time overhead is small, see Section 6.

Once a project has reached a stage, where its file structure has stabilized, the user can save time by setting a flag which prevents MAKE IT! from searching for new web or implementation files. The extra storage needed by MAKE IT! is neglectable.

Locality. To save time, MAKE IT! only considers dependencies to files within the project. Libraries of software packages, bibliography databases or LaTeX style files do not tend to change very frequently, so it is not necessary to make targets depend on them and to check this dependency in each call to `make`. The user can circumvent this restriction by setting links to external files.

Tidyness. MAKE IT! has an automatical "clean-up" service. If the user renames or removes an implementation or web file, MAKE IT! successively offers to delete every single file which stems from the corresponding source.

3 Initialization

In order to use MAKE IT!, a potential user just has to get and unpack a small tar file from the MAKE IT! homepage[1], include three lines into his shell set-up file, go to his current project directory, create a web or implementation directory, edit a file there, and then type `make`.

It was one of our goals to ask the user for nothing more than a simple call of `make` in his project directory in order to have all products and documentation of the project updated. Therefore we had to exploit a feature of GNU `make`, namely the functionality of the environment variable `MAKEFILES`. This variable tells `make` where to look for makefiles before considering the ones in the current directory. One of the three lines the user has to include into his shell set-up file sets this variable to the set-up makefile which is part of the MAKE IT! package. During the set-up, a standardized makefile is copied into the user's project directory and thus becomes the *project makefile*. Then three default dependency files are copied into the *dependency directory*, a subdirectory of the project directory where all dependency files are assembled, see appendix. After this set-up, the new project makefile takes over the control and includes the default dependency files.

MAKE IT! generates the inter-file dependency information in three main steps. Each of them is induced by one of the default dependency files. First MAKE IT! writes a dependency file for each web file, then it extracts all product files from the web files, and finally it writes the dependency files of all implementation files (without template code). To synchronize these steps, MAKE IT! uses flags and the order in which the default dependency files are included. It is essential that all targets of one step are completed before a target of a later step is made since each step needs information accumulated by its predecessor.

The difficulty we had to cope with was the following. On the one hand we had to convince `make` to read in all new web dependency files after our scan of the web directory and to include all implementation dependency files after we scanned the implementation directories. On the other

[1] The MAKE IT! homepage is `http://www.inf.fu-berlin.de/MakeIt`

hand we wanted `make` to do the goal targets only after updating all dependencies.

We could not use recursive calls of `make`. The information accumulated in the recursion would not have been available on the top level where the goal targets are finally processed. Instead, we use the fact that `make` tries to update all parts of which the makefile consists before dealing with goal targets. We wrote the commands to scan the web and the implementation directories into the default dependency files `webscan.dep` and `impscan.dep`, respectively. In case the web scan, say, discovers at least one web file, it touches[2] `webscan.dep` after writing dependency files for all web files. This causes `make` to restart itself and read in all the newly written dependency files – without forgetting the goal targets. To avoid that a scan is done twice, we use flags.

Apart from the set-up, this three-step initialization is repeated every time the user calls `make` – except a flag is set to switch it off explicitly to save time. Of course, dependency files are only written for files which are new or have been modified.

4 Making the Implementation

In this section we explain how we have solved the problem of extracting product files from a web file, how we find out which of the resulting files have to be compiled to object files, and how we finally link these together to generate executables.

Product File Generation. For the users of ANYWEB, this is a two-step process. First, an ANYWEB file has to be translated into a FUNNELWEB file from which FUNNELWEB then extracts the product files. Every FUNNELWEB (include) file just depends on the corresponding ANYWEB file.

The second step, the extraction of product files, is called "tangling" in the language of literate programming. Here, the dependencies between two groups of files have to be considered. One group contains the FUNNELWEB file and all the web files it includes, the other group consists of the product files. To ensure that the product files are always up to date, we must make each element of the second group depend on each element of the first. (Although we disallow a web include file to have its own product files, it can define a macro which the main web file writes into product files.)

This formulation causes two problems. Firstly, checking whether any of the product files has to be re-made takes quadratic time. Secondly and even worse, `make` will first find out for which product files the rule applies – say all – and then tangle the web file for each of them. `make` will not notice that they have been updated after the first tangling and thus do a lot of unnecessary work. For this reason, we introduced an artificial target as a mediator between the two groups of files. All product files of the web file depend on the artificial file which again depends on the web file and the web include files. (A graph theoretist would say that we replaced a complete bipartite graph by a star.) We determine the elements of the two groups by scanning the FUNNELWEB file, and then write the appropriate dependencies into its dependency file.

Compilation. The creation of an object file depends on its implementation file and on all header files which this implementation file includes. We use the preprocessor to determine these dependencies, then filter out all those which refer to files outside the project and write the remaining ones into the appropriate implementation dependency file. If any of the files, on which the object file depends, is more recent than the object file, the compiler must be called with a

[2]The UNIX command `touch` changes the modification time of a file to actual time.

list of include directories. We make this list by adding to the list of all external (user-specified) include directories all those directories, into which header files are tangled.

Linkage. If there is a product file whose name only occurs with an implementation file extension (`.C` by default), then MAKE IT! assumes that the user wants to create an executable of the same name, but without extension. In order to create this executable, all necessary object files must be linked together. MAKE IT! finds these by scanning the implementation file for the names of header files which belong to the project. For each of them, MAKE IT! has to check whether there is a corresponding implementation file. If this is the case, MAKE IT! assumes that they form a module and adds the corresponding object file to the list of files on which the executable depends.

Unfortunately, the header and the implementation file of a modul might have been stored in different subdirectories of the project directory. Since we wanted to avoid quadratic cost for detecting all pairs of header and implementation files with matching names, we came up with the following approach.

When we scan the implementation directories (see Section 3), we make a list of all implementation files. Then we take the list of all include directories (refer to "Compilation"), concatenate the two lists and sort the result. After removing duplicates from this list, we know that every consecutive pair of header and implementation file forms a module. For each such pair, we create an empty file of that name in the temporary files' directory. This is a witness for the fact that the header file belongs to a module, and we can find this witness more easily than the corresponding implementation file when writing the dependencies for an executable. The witness is unique since we disallow files with equal names within one and the same project.

5 Making the Documentation

Before a postscript file with the documentation is ready to print, the following steps have to be undertaken. Again, for ANYWEB users, first the ANYWEB file has to be translated into a FUNNELWEB file. From there, a latex file can be extracted or "woven" as literate programmers say. This is not the same FUNNELWEB file as the one for tangling. When tangling, ANYWEB puts emphasis on nicely setting the program code; when weaving ANYWEB takes care of extra commands like `@documentclass` and `@usepackage`, and moves them to the correct position in the FUNNELWEB file, see [Ket96]. FUNNELWEB only "knows" TeX, not LaTeX.

Within the documentation directory, a latex file is woven from such a FUNNELWEB file. Analogously to tangling, weaving depends on the web file and all files it includes. The difference is that we only have one output file here – the latex file – and thus do not need a mediator.

If this latex file has a `\bibliography` statement, BIBTeX must be executed after the first LaTeX run to generate a bibliography file. If the `\bibliography` statement contains references to bibliography databases within the project, MAKE IT! understands that the bibliography file depends on these local databases.

If the user includes other tex files or pictures into a latex file and informs MAKE IT! about the names of the include commands (like `\input` or `\epsfig`) that he uses, then MAKE IT! will call LaTeX whenever one of the included files changes.

The last step is purely generic: a postscript file is generated from, and only depends on the corresponding dvi (device independent) file which LaTeX produced.

6 Experiments

We tested Make It! on several projects, five of which are described here. Table 1 shows the size of each project, i.e. the number of web files, web include files, header files, implementation files, and the number of lines of code. The first two projects, DEB and SOC, are quite small, while the other three projects have some thousand lines of code. MAP and OPT use literate programming.

project	DEB	SOC	GEL	MAP	OPT
header files	4	14	17	10	20
implementation files	5	10	4	9	9
Anyweb files				10	5
Anyweb include files				2	1
lines of code	429	702	2846	2432	4026

Table 1: Size of projects.

Table 2 lists the number of targets and dependencies of the same projects, split into implementation and documentation. The number of implementation targets can only be compared among the first three projects and among the last two projects since the latter two have product files as additional implementation targets. All projects have a considerable number of implementation dependencies which must be found and maintained.

project		DEB	SOC	GEL	MAP	OPT
implementation	targets	6	10	8	36	31
	dependencies	24	53	46	79	83
documentation	targets				36	17
	dependencies				51	30

Table 2: Number of targets and dependencies.

Table 3 shows the execution times of different calls to `make`. All times are CPU seconds on a Sun SPARCstation-20 under SunOS-5.5 using the compiler SunPRO CC-4.2. The first row contains the times for the initial call of `make` which sets up Make It!, generates all dependency files and makes all implementation and documentation targets. In row 2 to 4, Make It! was already set-up, but the implementation and/or documentation targets, respectively, were removed before the call of `make`, i.e. no dependency files had to be generated. Row 5 and 6 show the times for calls of `make` after touching all implementation files. The calls of `make` and `make nop`,

project	DEB	SOC	GEL	MAP	OPT
`make` (from scratch)	45	73	133	317	307
`make` (after removing all targets)	23	34	102	252	243
`make` (after removing impl. targets)	23	34	102	140	101
`make` (after removing doc. targets)				112	150
`make` (after touching impl. files)	38	62	128	158	131
`make nop` (after touching impl. files)	15	28	26	32	34
`make rescan` (after `make`)	5	9	4	5	8

Table 3: Execution times in CPU seconds.

respectively, generated all implementation dependency files. In addition, the first call made all implementation targets. The last row contains the times for rescanning the web directory and the implementation directories for new files.

Table 4 compares the time needed for generating dependency files to that required for making targets. The first ratio quantifies the overhead for setting up MAKE IT!, i.e. for scanning directories and generating *all* dependency files, compared to the time for making all targets. This number is relatively large for small projects since they can be compiled and linked fast. The overhead for the remaining three projects is around 30%, but even this is not much more than one minute. Note that the time required for an average update will be far smaller since only the dependency files whose source changes will be re-written.

project	DEB	SOC	GEL	MAP	OPT
make all implementation and documentation targets	23	34	102	252	243
scan directories and generate all dependency files	22	39	32	65	64
ratio for setting up MAKE IT!	0.96	1.15	0.31	0.26	0.26
make all implementation targets	23	34	102	126	97
generate all implementation dependency files	15	28	26	32	34
ratio for making implementation targets	0.65	0.82	0.25	0.25	0.35

Table 4: Ratio of times required for generating dependency files and making targets.

The second ratio compares the time for generating dependency files to that for making only the targets for the implementation part of a project. Since no documentation is involved, these numbers can be compared among all projects. Again the overhead is around 30%, except for the two small projects. However, their second ratio is much smaller than their first.

Thus for small projects, one can cut down the overhead of MAKE IT! considerably by switching off the directory scan. For big projects, the overhead caused by the scan is neglectable. For all projects the overhead can be reduced to zero by setting a flag which tells MAKE IT! not to change dependency files any more. This makes sense when the file structure of a project has stabilized.

References

[BV88] Frank Bomarius and Matthias Vettermann. mmf, ein *makefile*-generator. Technical report, Fachbereich Informatik, Universität Kaiserslautern, 177/1988.

[FO96] John L. Furlani and Peter W. Osel. Abstract yourself with modules. In *10th Systems Administration Conference LISA 1996, Chicago, USA*, pages 193–203. USENIX, 1996.

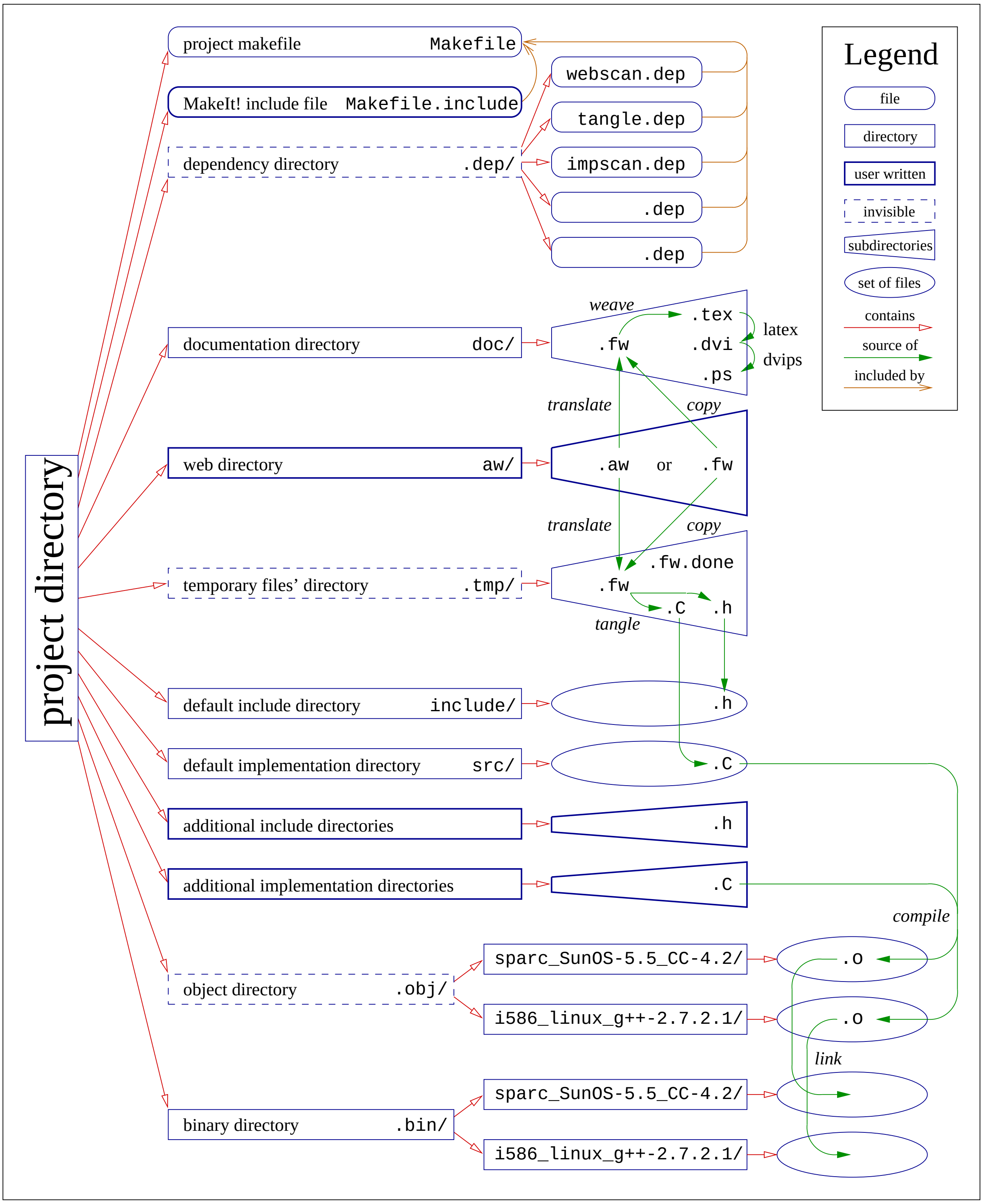
[Ket96] Lutz Kettner. *Anyweb.* `http://www.inf.ethz.ch/personal/kettner/others.html`, version 1.12 edition, November 1996.

[Knu84] Donald E. Knuth. Literate programming. *The Comp. Jour.*, 27(2):97–111, May 1984.

[SM94] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation.* Free Software Foundation, Inc., 0.47 edition, November 1994.

[Wil92] Ross Williams. Funnelweb user's manual. `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Australia, 1992.

Appendix



The directory structure used by MAKE IT!