Happy Manual

Happy version 0.9

February 1996

Andy Gill and Simon Marlow

1 Introduction

Happy is a parser generator system for Haskell, similar to the tool `yacc` for C. Like `yacc`, it takes a file containing an annotated BNF specification of a grammar and produces a Haskell module containing a parser for the grammar.

Happy is flexible; unlike `yacc`, you can have several Happy parsers in the same program. Happy can work in conjunction with a lexical analyser supplied by the user (either hand-written or generated by another program), or it can parse a stream of characters directly (but this isn't practical in most cases). In a future version we hope to include a lexical analyser generator with Happy as a single package.

Happy can currently generate three types of parser from a given grammar, the intention being that we can experiment with different kinds of functional code to see which is the best, and compiler writers can use the different types of parser to tune their compilers. The types of parser supported are:

1. 'standard' Haskell (should work with any compiler that compiles the Haskell language)
2. standard Haskell using arrays (this is not the default because we have found this generates slower parsers than 1).
3. Haskell with ghc (Glasgow Haskell) extensions. This is a slightly faster option than 1 for Glasgow Haskell users.

Parser type 1 will also work with Gofer, as will parser type 2 given a suitable implementation of arrays.

1.1 Compatibility

Happy is written in Haskell 1.2. This means that it will compile under Glasgow Haskell (recommended), hbc or (with a suitable prelude) Mark Jones' gofer interpreter. Unfortunately Happy will not compile under Yale Haskell, because Yale Haskell does not currently implement stream IO. With the release of the 1.3 Haskell standard, this incompatabilty problem, which is outwith our control, will go away. There is also an nhc port of Happy underway.

Remember: parsers produced using Happy should compile happily (sic) under any Haskell complier.

2 Obtaining Happy

The primary location of Happy is

```
ftp://ftp.dcs.gla.ac.uk/pub/haskell/happy
```

The following sites mirror this directory:

```
ftp://src.doc.ic.ac.uk/computing/programming/languages/haskell/happy
ftp://cs.chalmers.se/pub/haskell/(somehwere)
ftp://cs.yale.edu/pub/haskell/(somewhere)
```

Happy is also distributed along with the Glasgow Haskell Compiler. A World Wide Web page describing Happy can be found at

```
http://www.dcs.gla.ac.uk/fp/software/happy.html
```

3 Using Happy

Using Happy is much like using Yacc and the syntax is much the same. We allow use of the inverse comment notation (every line beginning with '>' is code), and using it we will go through a complete example.

Now let's examine a typical grammar file. The first thing to declare is the name of your parser, and the type of the tokens the parser reads.

```
> %name calc
> %tokentype { Token }
```

The parser will be of type `[Token] -> ?`, where `?` is determined by the production rules. Now we declare all the possible tokens:

```
> %token
>       let             { TokenLet }
>       in              { TokenIn }
>       int             { TokenInt $$ }
>       var             { TokenVar $$ }
>       '='             { TokenEq }
>       '+'             { TokenPlus }
>       '-'             { TokenMinus }
>       '*'             { TokenTimes }
>       '/'             { TokenDiv }
>       '('             { TokenOB }
>       ')'             { TokenCB }
```

The symbols on the left are the tokens as they will be referred to in the rest of the grammar, and to the right of each token enclosed in braces is the Haskell object that represents the token. The parser will expect to receive a stream of tokens, each of which will match one of the given patterns (the definition of the token datatype is given later).

The `$$` means that this terminal returns this component of the token, rather than the whole token. This saves the user writing a new projection function for each terminal.

Like yacc, we include `%%` here, for no real reason.

```
> %%
```

Now we have the production rules.

```
> Exp    : let var '=' Exp in Exp  { Let $2 $4 $6 }
>        | Exp1                    { Exp1 $1 }
>
> Exp1   : Exp1 '+' Term           { Plus $1 $3 }
>        | Exp1 '-' Term           { Minus $1 $3 }
>        | Term                    { Term $1 }
>
> Term   : Term '*' Factor         { Times $1 $3 }
>        | Term '/' Factor         { Div $1 $3 }
>        | Factor                  { Factor $1 }
>
> Factor
>        : int                     { Int $1 }
>        | var                     { Var $1 }
>        | '(' Exp ')'             { Brack $2 }
```

We are simply returning the parsed data structure! Now we need some extra code, to support this parser, and make it complete. The opening brace denotes a section of Haskell code to be included verbatim in the generated parser module. The code section is optional, and must be placed at the end of the grammar file. Another optional code section may be placed at the beginning of the file (this usually contains the module header).

```
> {
```

All parsers must declare the function `happyError`, which is called when an error is detected. Note that currently Happy cannot perform error recovery.

```
> happyError :: Int -> [Token] -> a
> happyError i _ = error ("Parse error in line " ++ show i ++ "\n")
```

Now we declare the abstract syntax that we are parsing.

```
> data Exp
>        = Let String Exp Exp
>        | Exp1 Exp1
>
> data Exp1
>        = Plus Exp1 Term
>        | Minus Exp1 Term
>        | Term Term
>
> data Term
```

```
>       = Times Term Factor
>       | Div Term Factor
>       | Factor Factor
>
> data Factor
>       = Int Int
>       | Var String
>       | Brack Exp
```

The data structure for the tokens...

```
> data Token
>       = TokenLet
>       | TokenIn
>       | TokenInt Int
>       | TokenVar String
>       | TokenEq
>       | TokenPlus
>       | TokenMinus
>       | TokenTimes
>       | TokenDiv
>       | TokenOB
>       | TokenCB
```

.. and a simple lexer that returns this data structure.

```
> lexer :: String -> [Token]
> lexer [] = []
> lexer (c:cs)
>       | isSpace c = lexer cs
>       | isAlpha c = lexVar (c:cs)
>       | isDigit c = lexNum (c:cs)
> lexer ('=':cs) = TokenEq : lexer cs
> lexer ('+':cs) = TokenPlus : lexer cs
> lexer ('-':cs) = TokenMinus : lexer cs
> lexer ('*':cs) = TokenTimes : lexer cs
> lexer ('/':cs) = TokenDiv : lexer cs
> lexer ('(':cs) = TokenOB : lexer cs
> lexer (')':cs) = TokenCB : lexer cs
>
> lexNum cs = TokenInt (read num) : lexer rest
>       where (num,rest) = span isDigit cs
>
> lexVar cs =
>    case span isAlpha cs of
>       ("let",rest) -> TokenLet : lexer rest
```

```
>        ("in",rest)  -> TokenIn : lexer rest
>        (var,rest)   -> TokenVar var : lexer rest
```

To run the program, call this in gofer, or use some code to print it.

```
> runCalc :: String -> Exp
> runCalc = calc . lexer
> }
```

And that it! A whole lexer, parser and grammar in a few dozen lines. Another good example is Happy's own parser. Several features in Happy were developed using this as an example.

To generate the Haskell module for this parser, type the command `happy example.ly` (where `example.ly` is the name of the grammar file). The Haskell module will be placed in a file named `example.hs`. Additionally, invoking the command `happy example.ly -i` will produce the file `example.info` which contains detailed information about the parser, including states and reduction rules. This can be invaluable for debugging parsers, but requires some knowledge of the operation of a shift-reduce parser.

4 Invoking

An invocation of Happy has the following syntax:

```
happy [ options ] <filename> [ options ]
```

All the command line options are optional (!) and may occur either before or after the input file name. There are quite a few options, but you will only need perhaps one or two on a regular basis.

There are two types of grammar files, file.y and file.ly, with the latter observing the reverse comment bird track convention. The examples distributed with Happy are all of the .ly form.

Caveat: When using hbc (Chalmers Haskell) the command argument structure is slightly different. This is because the hbc run time system takes some flags as its own (for setting things like the heap size, etc). This problem can be circumvented by adding a single `-` to your command line. So when using a hbc generated version of Happy, the argument structure is:

```
happy - [ options ] <filename> [ options ]
```

Pedantic programmers can either use aliases or a short shell wrapper to automatically add the `-`, along with any required run time system arguments (Or just use the Glasgow Haskell compiler to compile Happy!).

4.1 User Options

'`-v`'

'`--verbose`'
: Causes Happy to be more verbose. At the moment this just prints out the copyright message and has no effect on anything else.

'`-o <filename>`'

'`-outfile <filename>`'
: Specifies the destination of the generated parser module. If omitted, the parser will be placed in `<file>.hs`, where `<file>` is the name of the input file with any extension removed.

'`-i [ <filename> ]`'

'`--info [ <filename> ]`'
: Directs Happy to produce an info file containing detailed information about the grammar, parser states, parser actions, and conflicts. Info files are vital during the debugging of grammars. The filename argument is optional, and if omitted the info file will be written to `<file>.info` (where `<file>` is the input file name with any extension removed). Because this filename is optional, watch out for:

```
happy --info foo.ly
```

which will use foo.ly as the name of the information file, and complain that you have not specified a input file name. This can be solved by using:

```
happy foo.ly --info
```

'`-g`'

'`--ghc`'
: Instructs Happy to generate a parser that uses ghc-specific extensions to obtain faster code. When compiling the resulting Haskell module, remember to use ghc with the `-fglasgow-exts` option.

'`-a`'

'`--arrays`'
: Instructs Happy to generate a parser using an array-based shift reduce parser. Currently this generates slower parsers for two reasons: standard Haskell arrays are surrounded by overloading and bounds-checking which makes lookup quite expensive, and secondly current compilers don't know how to generate static objects (i.e. they generate the code which builds the array at run-time).

'`--template directory-name`'
: Instructs Happy to use this directory when looking for the template to including in the output.

Note: only one of the options `-g` and `-a` may be given (or their long versions). There is no array-based parser with ghc extensions yet, but this is planned for a future version.

4.2 Debugging Options

You won't need these, and they might not even work properly.

5 Grammar Files

The input to Happy is a text file containing the grammar of the language you want to parse, together with some annotations that help the parser generator make a legal Haskell module that can be included in your program.

The overall format of the grammar file is given below:

```
<optional module header>
<directives>
%%
<grammar>
<optional module trailer>
```

If the name of the grammar file ends in `.ly`, then it is assumed to be a literate script. All lines except those beginning with a `>` will be ignored, and the `>` will be stripped from the beginning of all the code lines. There must be a blank line between each code section (lines beginning with `>`) and comment section. Grammars not using the literate notation must be in a file with the `.y` suffix.

5.1 Lexical Rules

Identifiers in Happy grammar files must take the following form (using the BNF syntax from the Haskell Report):

```
id      ::= alpha { idchar }
          | ' { any{^'} | \' } '
          | " { any{^"} | \" } "

alpha   ::= A | B | ... | Z
          | a | b | ... | z

idchar  ::= alpha
          | 0 | 1 | ... | 9
          | _
```

5.2 Module Header

This section is optional, but if included takes the following form:

```
{
<Haskell module header>
}
```

The Haskell module header contains the module name, exports, and imports. No other code is allowed in the header—this is because Happy may need to include its own `import` statements directly after the user defined header.

5.3 Directives

This section contains a number of lines of the form:

```
%<directive name> <argument> ...
```

The statements here are all annotations to help Happy generate the Haskell code for the grammar. Some of them are optional, and some of them are required.

5.3.1 Token Type

```
%tokentype   { <valid Haskell type> }
```

This directive is mandatory, and gives the type of the tokens passed from the lexical analyser to the parser (in order that Happy can supply types for functions and data in the generated parser). For example, if the token type given in the `%tokentype` is `Char`, then the top level parsing function will have type `[Char] -> <result>` (where result is the result type of the parse, we'll see later how to specify this), and the parser will parse a string.

5.3.2 Tokens

```
%token <name> { <Haskell object> }
       <name> { <Haskell object> }
       ...
```

The `%token` directive is used to tell Happy about all the terminal symbols used in the grammar. Each terminal has a name, by which it is referred to in the grammar itself, and a Haskell representation enclosed in braces. Each of the objects must be of the same type, given by the `%tokentype` directive.

The name of each terminal follows the lexical rules for Happy identifiers given below. There are no lexical differences between terminals and non-terminals in the grammar, so it is recommended that you stick to a convention; for example using upper case letters for terminals and lower case for non-terminals, or vice-versa.

Happy will give you a warning if you try to use the same identifier both as a non-terminal and a terminal, or introduce an identifier which is declared as neither.

To save writing lots of projection functions that map tokens to their components, you can include `$$` in your Haskell pattern. For example:

```
%token <name> { TokenInt $$ }
        ...
```

This make the `<name>` rever to the first argument of `TokenInt` rather than the Token itself, eliminating the need for any projection function.

5.3.3 Parser Name

```
%name <Haskell identifier>
```

The `%name` directive is followed by a valid Haskell identifier, and gives the name of the top-level parsing function in the generated parser. This is the only function that needs to be exported from a parser module.

The parser name directive is optional, and, if omitted, defaults to `happyParse`.

5.3.4 Newline Token

```
%newline { <Haskell object> }
```

Optionally, you may designate one token as a *newline* token. The object given as a newline token should be of the type given by the `%tokentype` directive.

Happy maintains a current line count based on the number of newline tokens occurring in the input stream. This line number is passed to all reduction functions, so that if necessary you can include line number information in your abstract syntax for error reporting purposes.

For example, if your parser is parsing a string directly, the newline token will be `\n`:

```
%newline { '\n' }
```

Beware: This feature may disappear in a future version of Happy to make way for a more general user defined 'state' object. The state may of course contain line number information. It should also let you write parsers involving, for example, the Haskell offside rule.

5.3.5 Monadic Parsers

```
%monad { <type> } { <then> } { <return> }
```

Happy can help when the parser is to be within a monad, by generating much of the code automatically. The `%monad` directive takes three arguments: the type constructor of the monad, the `then` (or `bind`) operation, and the `return` (or `unit`) operation.

For example, if we have a standard state monad defined as follows:

```
> type S a = State -> (a,State)
> (m `thenS` k) s = let (a,s') = m s in k a s'
> returnS m s = (m,s)
```

The following directive can be included in the grammar file:

```
%monad { S } { thenS } { returnS }
```

Happy will then translate productions of the following form

```
> f : t_1 ... t_n { e }
```

into

```
> f : t_1 ... t_n { $1 `thenS` ... `thenS` $n `thenS` (returnS e) }
```

Currently, the lexer is not threaded into the monad, but this is planned as a future extension.

5.4 Grammar

The grammar section comes after the directives, separated from them by a double-percent (%%) symbol. This section contains a number of *productions*, each of which defines a single non-terminal. Each production has the following syntax:

```
<non-terminal> [ :: { <type> } ]
        :  <id> ... { <expression> }
       [ |  <id> ... { <expression> }
         ... ]
```

The first line gives the non-terminal to be defined by the production and optionally its type. There are a few advantages to providing types for non-terminals: they can reduce compilation time for the generated parser, finding type errors can be easier, and they provide a certain amount of documentation.

Each production has at least one, and possibly many right-hand sides. Each right-hand side consists of zero or more symbols (terminals or non-terminals) and a Haskell expression enclosed in braces.

The generated parser will attempt to match sequences of symbols on the input against the right-hand sides of the non-terminals. When a match is found, a *reduce* operation is performed. This involves evaluating the expression associated with the right-hand side of the matching non-terminal. The expression refers to the semantic values of the symbols in the matched sequence using the meta-variables $1 .. $n. The meta-variable $# refers to the current line-number.

Remember that all the expressions for a production must have the same type.

For example, a production like the following is often used for parsing lists:

```
intlist :: { [Int] }
        : intlist int           { $2 : $1 }
        |                       { [] }
```

This example demonstrates a couple of useful points:

Although Happy will correctly parse rules involving both left and right recursion, in general it is better to use left recursion. This is because left-recursive rules consume constant parser stack space, whereas right-recursive rules need stack space linear in the length of the sequence being parsed. Unfortunately, using left recursion can be a pain because the resulting list comes out in reverse order, and may need to be reversed again later.

It is perfectly legal to have a right hand side with no symbols (the second line in the `intlist` production above), and this really does match a sequence of no symbols. Be careful, however, as too many empty rules can be a source of conflicts (ambiguities) in the grammar.)

5.5 Module Trailer

The module trailer is optional, comes right at the end of the grammar file, and takes the same form as the module header:

```
{
<Haskell code>
}
```

This section is used for placing auxiliary definitions that need to be in the same module as the parser. In small parsers, it often contains a hand-written lexical analyser too. There is no restriction on what can be placed in the module trailer, and any code in there is copied verbatim into the generated parser file.

6 Lexical Analysers

This page intentionally left blank.

7 Tips

7.1 Performance Tips

If you are using ghc, generate parsers using the `-g` option, and compile them using ghc with the `-fglasgow-exts` option.

The lexical analyser is usually the most performance critical part of a parser, so it's worth spending some time optimising this. Profiling tools are essential here.

Simplify the grammar as much as possible, as this reduces the number of states and reduction rules that need to be applied.

Use left recursion rather than right recursion wherever possible. While not strictly a performance issue, this affects the size of the parser stack, which is kept on the heap and thus needs to be garbage collected.

7.2 Compilation Time Tips

We have found that compiling parsers generated by Happy can take a large amount of time/memory, so here's some tips on making things more sensible:

Include as little code as possible in the module trailer. This code is included verbatim in the generated parser, so if any of it can go in a separate module, do so.

Give type signatures for everything. This is reported to improve things by about 50%. If there is a type signature for every single non-terminal in the grammar, then Happy automatically generates type signatures for most functions in the parser.

Simplify the grammar as much as possible (applies to everything, this one).

7.3 Finding Errors

Finding errors in grammar files is inherently difficult because the code for reductions is moved around before being placed in the parser. We currently have no way of passing the original filename and line numbers to the Haskell compiler, so there is no alternative but to look at the parser and match the code to the grammar file. An info file (generated by the `-i` option) can be helpful here.

(this section will contain more info as we gain experience with creating grammar files. Please send us any helpful tips you find.)

7.4 Conflict Tips

Conflicts arise from ambiguities in the grammar. That is, some input sequences may possess more than one parse. Shift/reduce conflicts are benign in the sense that they are easily resolved (Happy automatically selects the shift action, as this is usually the intended one). Reduce/reduce conflicts are more serious. A reduce/reduce conflict implies that a certain sequence of tokens on the input can represent more than one non-terminal, and the parser is uncertain as to which reduction rule to use. It will select the reduction rule uppermost in the grammar file, so if you really must have a reduce/reduce conflict you can select which rule will be used by putting it first in your grammar file.

It is usually possible to remove conflicts from the grammar, but sometimes this is at the expense of clarity and simplicity. Here is a cut-down example from the grammar of Haskell (1.2):

```
exp     : exp op exp0
        | exp0

exp0    : if exp then exp else exp
        ...
        | atom

atom    : var
        | integer
        | '(' exp ')'
        ...
```

This grammar has a shift/reduce conflict, due to the following ambiguity. In an input such as

```
if 1 then 2 else 3 + 4
```

the grammar doesn't specify whether the parse should be

```
if 1 then 2 else (3 + 4)
```

or

```
(if 1 then 2 else 3) + 4
```

and the ambiguity shows up as a shift/reduce conflict on reading the 'op' symbol. In this case, the first parse is the intended one (the 'longest parse' rule), which corresponds to the shift action. Removing this conflict relies on noticing that the expression on the left-hand side of an infix operator can't be an '`exp0`' (the grammar previously said otherwise, but since the conflict was resolved as shift, this parse was not allowed). We can reformulate the '`exp`' rule as:

```
exp     : atom op exp
        | exp0
```

and this removes the conflict, but at the expense of some stack space while parsing (we turned a left-recursion into a right-recursion). There are alternatives using left-recursion, but they all involve adding extra states to the parser, so most programmers will prefer to keep the conflict in favour of a clearer and more efficient parser.

7.4.1 LALR(1) parsers

There are three basic ways to build a shift-reduce parser. Full LR(1) (the 'L' is the direction in which the input is scanned, the 'R' is the way in which the parse is built, and the '1' is the number of tokens of lookahead) generates a parser with many states, and is therefore large and slow. SLR(1) (simple LR(1)) is a cut-down version of LR(1) which generates parsers with roughly one-tenth as many states, but lacks the power to parse many grammars (it finds conflicts in grammars which have none under LR(1)).

LALR(1) (look-ahead LR(1)), the method used by Happy and '`yacc`', is tradeoff between the two. An LALR(1) parser has the same number of states as an SLR(1) parser, but it uses a more complex method to calculate the lookahead tokens that are valid at each point, and resolves many of the conflicts that SLR(1) finds. However, there may still be conflicts in an LALR(1) parser that wouldn't be there with full LR(1).

8 Future

Happy is a dynamic project. There are several shortcomings of this version that will be rectified. Here are some notes about ideas that will be explored in the future. Comments are most welcome.

1. Lexical Analysis

   The lack of a lexer is a shortcoming, even though writing lexers in Haskell is a lot easier than writing them in C. There are going to be two main advances in future Happies. First a way of specifying a regular expression as a token match, as well as the current scheme of pattern matching. Secondly a way of explaining to Happy the actual structure of the lexical objects, which will allow for further optimisations.
2. Connectivity between Lexer and Parser

   This will allow parsers to cope with strange grammars, like the Haskell grammar with layout. Monads will probably be used for this.
3. Error recovery

   We intend to implement error recovery using the error token as in `yacc`.
4. Parser Debugging

   This feature will allow the user to trace the action of the parser, showing the current state and actions taken during parsing.
5. More parser types

   A planned parser type will be an array based parser using ghc extensions. These should be the smallest and fastest parsers available for ghc users. To do this, however, we may have to resort to using C arrays and Haskell functions to read these arrays.
6. Different parsing algorithms

   Recursive ascent, for instance.

Also see the file `TODO` in the distribution.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DE-

FECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Table of Contents