DGrammar

S. van Leent

October 7, 2005

Contents

Chapter 1

Preface

The idea behind DGrammar was to develop a Grammarcompiler, or sometimes also called a Compiler/Compiler, to be able to interpret a Grammar specification and compile it to a D file, to be used as a D package.

The first thought occured, how should this thing look like? Should it act as YACC, or should it do something completely different such as LALR. Well, to be honest, initialy the Syntax of DGrammar looks like YACC, but apart from that, many things are changed. DGrammar shares a lot of ideology with the de facto XML SAX standard.

But why would one develop a new Grammarcompiler. Well, YACC works, but it is aging, and it only works decently with C or C++ code. Besides that, YACC looks quite ugly. Execution code is mixed into the grammar file, which makes the grammar file rather difficult to read and even more of a hazzard to maintain.

Why for D? This is a valid question, and not a stupid one either. DGrammar could just as well be developed for C++ or Java. But the specific behaviour such as list-slicing, a fast runtime regular expression parser, and a very good working Garbage Collector makes D an excellent choice. Surely because when parsing, objects are created and later on abandonded because of better paths.

In this document, references are written like *reference*, text output is written like `output` and text input is written like ***`input`***. If it is needed to hit the Enter key, it is shown with: ↩.

Chapter 2

A first look

2.1 The famous example

These days many people like to write extensive introductions on how something works before actually working with it. In this book we just begin with writing the first application. Before diving into it, make sure you have a D compiler such as DMD, available from *http://www.digitalmars.com/d/* and DGrammar installed properly (Appendix A).

Let's look how a simple DGrammar file looks like:

```
%module example1
Body:
    "Hello World!"/s                      [ ParseBody ]       ;
```

The `%module` directive tells the DGrammar compiler that the compiler generated should be placed in the module 'example1'. It behaves similar to what D does with `module`.

2.2 The Grammar

The grammer itself shows some particular interesting information. The first line contains the `Body` statement, followed by a colon. This marks the beginning of a grammar, in this case the grammar named `Body`.

On the next line there is a statement `"Hello World!"/s`, which indicates that a string should be matched with the characters in that order. One can read this part as:

$\Sigma = \{Hello \sqcup world!\}$
$L_1 = \{Hello \sqcup World!\} \in \Sigma^*$

So the language L_1 contains the sentence 'Hello World!'. No other expressions are specified, what is left over is the `[ ParseBody ]` Rule specifier, which is used while evaluating, and the semicolon, ending the grammar.

The file could be saved as 'example1.d'.

2.3 Compile and Run

Before it is possible to actually use the grammar it needs to be compiled to a D file. This looks like the way YACC works, compiling it to C or C++ in that case. This has the advantage that the grammar is compiled and therefore is faster. To compile the grammar file, the following command needs to be executed in a console environment:

```
$ dgrammar example1.dg -o example1.d ↩
```

As can be seen, it looks all quite familiar. One gives a file, in this case 'example1.dg' and let dgrammer write it to 'example1.d'.

However, before it is possible to use the newly created parser, we need to create a simple D procedure invoking the parser. The following illustrates a simple method to do this:

```
module main;

import std.cstream;
import std.stdio;
import std.string;
import example1;

int main(char[][] args) {
    char[] line;
    line = (new CFile(stdin, FileMode.In)).readLine();
    line = strip(line);

    register();
    Parser parser = parse(EParser.Body, line);

    if(parser is null) {
        writefln("You didn't enter 'Hello World!'");
    } else {
        writefln("Hello to you!");
    }
    return 0;
}
```

First, characters are read from the 'stdin', or simply the command line. After the characters are read, the parser has to be initialized using the `register()` function. What remains is to parse the line, using the `parse()` function. The function expects to arguments, the initial Grammar, in this case there is only one grammar, `EParser.Body`, and the string which needs to be parsed.

If the resulting parser object, returned from the parse function is non-existing, e.g. `null`, then something went wrong, otherwise, the parse cycle was successful.

This file could be written to 'main.d', and after that the following rule needs to be executed when using dmd:

```
$ dmd main.d example1.d -ofexample1 ↩
```

After compiling the code, the last thing to happen is to execute the file:

```
$ ./example1 ↩
Hello World! ↩
Hello to you!
```

If this all works, the first compiler seems to be working. It doesn't do much, and it probably isn't very useful, but it works.

Chapter 3

A better grammar

3.1 Logical separation

Let's look whether it is possible to split the 'Hello World!' grammar into more logical parts. What to think of the parts 'Hello', '␣' and 'World!'? These parts can be divided into three sections. The following code represents a grammar file doing this:

```
%module example2
Body:
    "Hello"/s " "/s "World!"/s            [ ParseBody ]      ;
```

This is much more versatile, since the grammar now is devided into parts. Any language L_n can exists out of any sentence or sentences which can be created from Σ^*. Now, within the grammar `Body`, three Languages are specified:

$\Sigma = \{Hello, \sqcup, World!\}$
$L_1 = \{Hello\} \in \Sigma^*$
$L_2 = \{\ \sqcup\ \} \in \Sigma^*$
$L_3 = \{World!\} \in \Sigma^*$

Together they form a concatenation of the language: $L_p = L_1L_2L_3$. So L_p defines the language $L_p = \{Hello \sqcup World!\}$. Technically this grammar is not acting different as the previous grammar, but it shows how to logically split up the grammar in different components.

3.2 Case insensitive matching

Sometimes, you don't only want "Hello World!" to be valid, but also variations, such as "HELLO WORLD!" or "hello world!". This can be achieved using case insensitive matching. If we want to do this, we have to rewrite the previous example as:

```
%module example2a;
Body:
    "Hello"/si " "/s "World!"/si          [ ParseBody ]       ;
```

The `/si` switch indicates that a sentence matches against all case variants. This allows you to create a grammar used in programming languages as BASIC and ADA.

3.3 Whitespaces

Now it is somewhat cumbersome to have to specify the whitespace character within a specific symbol every time again. It makes the grammar difficult to read. So the following grammar exposes the `%ignore` directive. This directive gives us the possibility to to ignore a specified symbol.

```
%module example3
%ignore WhiteSpace
Body:
    "Hello"/s "World!"/s                  [ ParseBody ]      ;

<WhiteSpace> WhiteSpace:
    "[\s]+"/r                             [ ParseWhite ]     ;
```

The `%ignore` tells the generated parser that it has to skip everything being a whitespace. Well, not exactly. It tells the parser that it has to skip every whitespace within a specified machine. However, the standard machine is unnamed, and not visible. A machine is specified between `<` and `>`. These specifiers are within another scope as the symbol specifiers and therefore the specifier names may be the same, they don't clash.

As can be seen, the `WhiteSpace` grammar is defined within another machine. This is necessary, because if we need to skip whitespaces, we don't want to skip them also within the `WhiteSpace` symbol itself, because although it is a correct specification, the generated parser will crash.

When calling another machine, the symbol itself specifying the machine switches the context, so that is why the `%ignore` directive doesn't need to specify the name of the new machine. Developers with knowledge of the FLEX or LEX lexer will see something familiar here.

Another new thing is the `/r` switch, which indicates that we have to do with a regular expression instead of a normal string. Regular expressions are those used in the `std.regexp` package of D. Of course, it is also possible to use `/ri` which is, just as the string counterpart `/si`, case insensitive.

This grammar is different from the previous grammar because it allows to enter more spaces between 'Hello' and 'World!'. The following languages specify how the parser acts:

$L_{a1} = \{Hello\} \in \Sigma^*$
$L_{a2} = \{World!\} \in \Sigma^*$
$L_b = \left\{\{\sqcup\}^+\right\} \in \Sigma^*$
$L_p = L_b L_{a1} L_b L_{a2} L_b$

Now we can slightly alter the code of the D file, the D file using this code would now look like:

```
module main;

import std.cstream;
import std.stdio;
import example3;

int main(char[][] args) {
    char[] line;
    line = (new CFile(stdin, FileMode.In)).readLine();

    register();
    Parser parser = parse(EParser.Body, line);

    if(parser is null) {
        writefln("You didn't enter 'Hello World!'");
    } else {
        writefln("Hello to you!");
    }
    return 0;
}
```

Notice that the strip function isn't needed anymore, since it is automatically implemented with the whitespace grammar. When running the build executable, the following gives an impression of possibilities:

```
             $ ./example3 ↩
+------------------+-----------------------+
| Hello World! ↩   |     Hello   World! ↩  |
| Hello to you     | Hello to you          |
+------------------+-----------------------+
```

3.4 Colon or Arrow

The previous example can be rewritten using an arrow instead of a colon to identify each symbol. The following code performs the same task as the previous example:

```
%module example3a
```

```
%ignore WhiteSpace
Body -->
    "Hello"/s "World!"/s                  [ ParseBody ]      ;

<WhiteSpace> WhiteSpace -->
    "[\s]+"/r                             [ ParseWhite ]     ;
```

It is just a matter of what you like to choose. It is also fully legal to use them both in the same source, but that makes it much harder to understand, do be somewhat disciplined and abstain from doing that.

Chapter 4

Evaluation

4.1 Simple evaluation

The previous generated parsers didn't do much usefull, besides checking whether or not one did enter a string "Hello World!". But a real parser ofcourse is able to check logical parts. So the following grammars introduce a way to let the scanned line interoperate with the generated parser. It will choose to print a number or character on the screen, when one enters the keyword "print";

```
%module example4;
%ignore WhiteSpace;
Body:
    "print"/s "[0-9]+"/r                    [ PrintNum ]        |
    "print"/s "[a-zA-Z]+"/r                 [ PrintAlpha ]      ;

<WhiteSpace> WhiteSpace:
    "[\s]+"/r                               [ ParseWhite ]      ;
```

What's done is that the parser checks whether one entered "print" followed by a number or by character. The grammar can be read as:

$L_a = \{print\} \in \Sigma^*$
$L_{bn} = \left\{\{0,1,2,3,4,5,6,7,8,9\}^+\right\} \in \Sigma^*$
$L_{ba} = \left\{\{a,b,\ldots,z,A,B,\ldots,Z\}^+\right\} \in \Sigma^*$
$L_b = \left\{\{\sqcup\}^+\right\} \in \Sigma^*$
$L_{an} = L_b L_a L_b L_{bn} L_b$
$L_{aa} = L_b L_a L_b L_{ba} L_b$
$L_p = L_{an} \cup L_{aa}$

As one can see, describing such a parser without the help of a grammar and without regular expressions, it becomes much of an art which only a few want to do.

The generated parser can be used to create a compiler that does do something with the entered line. Previously, it didn't do more as just validating if the code was sane, but we can evaluate the code by creating an evaluator. To do this, one derives a class from the `Evaluator` interface. Here is how it is done:

```
module main;

import std.cstream;
import std.stdio;
import example4;

public class MyEvaluator : Evaluator {
    public Object go(
        EParser parserType, Rule rule, char[] ruleName) {
        if(parserType == EParser.Body &&
           ruleName == "PrintNum") {
            writefln("Number: %s",
                rule.elementAt(1).getMatch());
        } else {
            writefln("String: %s",
                rule.elementAt(1).getMatch());
        }
        return null;
    }
}

int main(char[][] args) {
    char[] line;
    line = (new CFile(stdin, FileMode.In)).readLine();

    register();
    Parser parser = parse(EParser.Body, line);

    if(parser is null) {
        writefln("You entered garbage");
    } else {
        writefln("It parsed, let's evaluate");
        evaluate(parser, new MyEvaluator());
    }
    return 0;
}
```

4.2 Exclusive matching

In many programming languages words are used to identify a specific keyword. For example, in C, “struct” is used to identify a structure or record of different types. However, words as “struc” and “structu” are variable names. If using a regular expression such as `[A-Za-z]*` it would match any of them. But the exclusive match, if found, overrides the regular expression, making it invalid.

The following, modified version of the previous grammar explicitly checks for the keyword “hello”. An example:

```
%module example4a;
%ignore WhiteSpace;
Body:
    "print"/s "[0-9]+"/r                    [ PrintNum ]       |
    "print"/s "[a-zA-Z]+"/r                 [ PrintAlpha ]     |
    "print"/s "hello"/sxi                   [ PrintHello ]     ;

<WhiteSpace> WhiteSpace:
    "[\s]+"/r                               [ ParseWhite ]     ;
```

In this case, any case insensitive word defining “hello” is automatically used apart from the others. To use this feature, use the switch `/sxi` or the case sensitive version `/sx`. The following D code shows how it can be used:

```
module main;

import std.cstream;
import std.stdio;
import example4a;

public class MyEvaluator : Evaluator {
public Object go(
        EParser parserType, Rule rule, char[] ruleName) {
if(parserType == EParser.Body &&
   ruleName == "PrintNum") {
writefln("Number: %s", rule.elementAt(1).getMatch());
} else if(parserType == EParser.Body &&
          ruleName == "PrintAlpha"){
writefln("String: %s", rule.elementAt(1).getMatch());
} else {
writefln("%s", "Hello to you!");
}
return null;
}
```

```
}

int main(char[][] args) {
char[] line;
line = (new CFile(stdin, FileMode.In)).readLine();

register();
Parser parser = parse(EParser.Body, line);

if(parser is null) {
writefln("You entered garbage");
} else {
writefln("It parsed, let's evaluate");
evaluate(parser, new MyEvaluator());
}
return 0;
}
```

When running the program, enter ***`print Hello`*** and it will return `Hello to` you back.

4.3 A Nice Grammar

To illustrate how the `DefaultEvaluator` works, the following dgrammar file defines a parser which actually does something useful. It calculates using the operators +, -, * and /, on signed integers.

```
%module example5;
%ignore WhiteSpace;

< WhiteSpace > WhiteSpace:
    "[\s\r\n\t]"                       [ WhiteSpace ]       ;

Declarations:
    Intermediate ";"/s Declarations      [ Decl ]             |
                                         [ Empty ]            ;
Intermediate:
    AddExp                               [ Intermediate ]     ;

AddExp:
    AddExp "+"/s MulExp                  [ Add ]              |
    AddExp "-"/s MulExp                  [ Substract ]        |
    MulExp                               [ MulExp ]           ;
```

```
MulExp:
    MulExp "*"/s Argument               [ Multiply ]         |
    MulExp "/"/s Argument               [ Divide ]           |
    Argument                            [ Argument ]         ;

Argument:
    "\-?[0-9]+"                         [ Number ]           ;
```

A closer look reveals that it is possible to call other grammars and to use recursion. Also, the WhiteSpace and Argument grammars define regular expressions. Note that it is not needed to use the /r switch for regular expressions.

4.4 A Nice Evaluator

The evaluator used to evaluate the generated parser is derived from the template class `DefaultEvaluator`. It supports a set of nice features such as the `set()` and `nullify()` methods.

The `nullify()` method skips the evaluation of a part of the tree. It needs one argument, being the position of the tree to skip. The `set()` method changes the evaluator type from one to another, so one is able to split up the evaluation in different classes, hence also in different modules, making it much easier to maintain. The `set()` method needs two arguments, the position of the tree and the new object to use as evaluator.

The following is an implementation of such an evaluator:

```
module main;

import example5;
import std.conv;
import std.stream;
import std.stdio;

public class Float {
    float number;
    this(float number) {
        this.number = number;
    }
}

public class MyEvaluator : DefaultEvaluator!(MyEvaluator) {
    private EParser currentParser;
    private char[] ruleName;
    private float finalNumber;
```

```
public void enter(
    EParser parserType, Rule rule, char[] ruleName) {

    this.currentParser = parserType;
    this.ruleName = ruleName;

    if(currentParser == EParser.Argument) {
        if(ruleName == "Number") {
            finalNumber =
                toInt(rule.elementAt(0).getMatch());
        }
        nullify(0);
    }

}

public Object leave(Object[] resultSet) {
    if(currentParser == EParser.Argument) {
        return new Float(finalNumber);
    } else if(
        ruleName == "Add" ||
        ruleName == "Substract" ||
        ruleName == "Multiply" ||
        ruleName == "Divide") {
        float number1 =
            (cast(Float)(resultSet[0])).number;
        float number2 =
            (cast(Float)(resultSet[2])).number;
        switch(ruleName) {
            case "Add":
                return new Float(number1 + number2);
            break;

            case "Substract":
                return new Float(number1 - number2);
            break;

            case "Multiply":
                return new Float(number1 * number2);
            break;

            case "Divide":
                return new Float(number1 / number2);
```

```
                break;
            }
        } else if(ruleName != "Empty") {
            if(ruleName == "Intermediate") {
                writefln(
                    "%s", (cast(Float)(resultSet[0])).number);
            }
            return resultSet[0];
        }
        return new Float(0);
    }
}


public int main(char[][] args) {
    if(args.length <= 1) {
        writefln("usage: %s <filename>", args[0]);
        return 1;
    }
    File f = new File(args[1]);
    char[] s = f.toString();

    register();

    Parser parser = parse(EParser.Declarations, s);
    if(parser is null) {
        writefln(
            "You entered garbage");
        return 1;
    }
    else {
        evaluate(parser, new MyEvaluator());
    }

    return 0;
}
```

This should be able to compile, and there you are, a nice little calculator, able to use a file with your calculations, split with a semicolon.

Chapter 5

Another Example

The examples included with the source of DGrammar include an example called "Easy". It is a kind of script language interpreter, which allows you to calculate on numbers, variables and includes different kinds of statements, whereof one statement is a while loop. The while loop stops when the number assigned to it reaches 0.

The example script file looks like this:

```
begin:

    declare var;
    assign var := 9 * 5 + 5 * -7;
    assign var := 10 * var;
    print var;

    while var begin:
        assign var := var - 1;
        print var;

        declare sub;
        assign sub := 3 * 10 + 3;
        print sub;

        while sub begin:
            assign sub := sub - 2;
            print sub;
        end;
    end;

    print var;
end;
```

It looks a bit like Pascal, but misses most of the features. It is a very simple language, but it illustrates the possibilities of DGrammar. As you can see, it allows you to declare variables and to assign variables. Variables can be (re-)declared within a scope, so they can be reused.

Appendix A

Installing DGrammar

Installing DGrammar requires some knowledge about make, and it is necessary to have FLEX, Bison, C, C++ and DMD. One also needs permission to install programs.

To Build DGrammar, it is necessary to go to the root directory of the DGrammar source folder. Then using make, enter:

```
$ make ↩
```

After making the program, use make to install:

```
$ make install ↩
```

This should install DGrammar.

Appendix B

GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.