GSL/4.1 - a Universal Code Generator

Brett Viren

October 14, 2019

Contents

1 Overview

1.1 Contributing

We use the C4.1 process, see: `https://rfc.zeromq.org/spec:22`.

1.2 Scope and Goals

GSL/4.1 is a code construction tool. It will generate code in all languages and for all purposes. If this sounds too good to be true, welcome to 1996, when we invented these techniques. Magic is simply technology that is twenty years ahead of its time. In addition to code construction, GSL has been used to generate database schema definitions, user interfaces, reports, system administration tools and much more.

This is the fourth major version of GSL, now considered a stable product, repackaged together with its dependencies for easy building from git.

1.3 Ownership and License

1.3.1 Ownership and License of generated sources

1.4 Building and installing

Dependencies:

- pcre package (e.g. libpcre3-dev)

To build from git on a UNIX-like box, and install into `/usr/local/bin`:

```
git clone git://github.com/zeromq/gsl
cd gsl/src
make
sudo make install
```

To install it to another location, change the last command to:

```
sudo make install DESTDIR=/my/custom/prefix
```

To show command-line help:

```
./gsl
```

1.4.1 Building on FreeBSD 10

Install GNU Make and GNU Compiler. For example, with `pkg`, `pkg install gmake gcc`. Then edit `src/Makefile` and add "-lm" to `src/Makefile` where you see CCLIBS configured. It may look similar to:

```
export CCLIBS = -lpcre
```

You want to add the math library:

```
export CCLIBS = -lpcre -lm
```

Cd to `src` and run:

```
CCNAME=gcc47 gmake
gmake install
```

1.4.2 Building on Cygwin

Install apt-cyg, a cygwin package manager:

```
lynx -source rawgit.com/transcode-open/apt-cyg/master/apt-cyg > apt-cyg
install apt-cyg /bin
```

Install git:

```
apt-cyg install git
```

Install gcc's dependencies:

```
apt-cyg install wget gcc-g++ make diffutils libmpfr-devel libgmp-devel libmpc-devel libpcre-devel libc
```

Download, Build and Install gcc:

```
wget http://ftpmirror.gnu.org/gcc/gcc-4.9.2/gcc-4.9.2.tar.gz
tar xf gcc-4.9.2.tar.gz
mkdir build-gcc && cd build-gcc
../gcc-4.9.2/configure --program-suffix=-4.9.2 --enable-languages=c,c++ --disable-bootstrap --disable-
make -j4
make install
```

Finally build gsl:

```
git clone git://github.com/zeromq/gsl
cd gsl/src
make
make install
```

1.4.3 Building on MacOS

The modern way of building on MacOS is to make sure you have pcre installed and use brew.

```
brew install pcre
```

And then build gsl as above:

```
git clone git://github.com/zeromq/gsl
cd gsl/src
make
sudo make install
```

1.5 This Document

This document was written by Pieter Hintjens in October 2010 based on two 2005 articles on 'model oriented programming', and the GSL reference manual. This text is originally at README.txt and is built using gitdown. The text was updated by Gyepi Sam in January 2013 to port documentation from earlier versions and to include more examples.

2 Starting with GSL

GSL is an acronym for Generator Scripting Language. And that is what it does. You write scripts in gsl, feed it some data from some XML files and it generates nicely formatted text files for you. These files can be source code, a web site, a recipe book or whatever you like.

Read on to get you started with code generation!

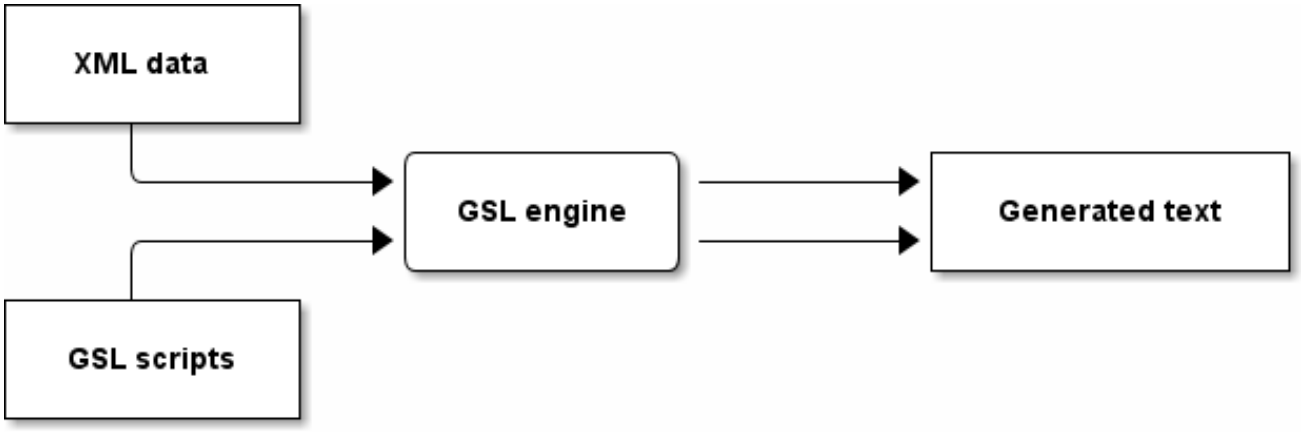

Figure 1 — General Process

2.1 Hello World

Our first step is to make a "hello world" program in GSL. It's quite simple. Make a file called `hello.gsl` that contains one line:

```
echo "hello world"
```

To run this, use the following command:

```
gsl hello
```

GSL is a simple language and you'll not have any difficulty understanding its syntax, except in a few places where it does specialised work. It will take you a little longer to understand what you can do with GSL, but that is the real point of these articles. GSL is not as rich as other scripting languages. It is a code generator scripting language, not a programming tool. It lacks some control structures, and it runs a little slowly.

Initially, GSL looks like any other scripting language. I can write little scripts like this:

```
amount = 1000
year = 2006
while year < 2026
    amount = amount * 1.05
    year = year + 1
endwhile
echo amount
```

Which calculates the value of my savings account if I were to leave it untouched for twenty years, and the interest rate were steady at five percent. Note these syntax aspects:

- `variable = expression` - Assign a value to a variable
- `while condition... endwhile` - Repeat a block while the condition is true

To run the above program, assuming it was saved in a file called `interest.gsl`, I type this command:

```
gsl interest
```

This executes the script and tells me that if I am really patient, I'll be rich one day. Now I'm going to change this little program to make the same kind of calculation for different amounts, rates, and years. Where do I put these different terms and rates? The answer is, in an XML file. The file is called `deposits.xml`:

```
<?xml version="1.0"?>
<deposits script = "interest.gsl" >
    <deposit amount = "1000000" rate = "5" years = "20" />
    <deposit amount = "500000" rate = "4" years = "10" />
    <deposit amount = "2500000" rate = "6" years = "15" />
</deposits>
```

We change our script to give the result below.

```
.template 0
 for deposit
    year = 1
    accumulated = amount
    while year < years
        accumulated = accumulated * (rate / 100 + 1)
        year = year + 1
    endwhile
    echo "Original amount:" + amount + " becomes: " + accumulated
 endfor
.endtemplate
```

Note these syntax aspects:

- `.template 0` - Start script (non-template) block
- `for <childname>` - Repeat block for all instances of child item called `childname`

We will run the new interest calculation script using this command:

```
gsl deposits.xml
```

Note the change of command syntax. We first ran the GSL script. Now we're running the XML file. This is one of GSL's features - you can run XML files as if they were scripts. It's the `script =` setting that does the trick, working much like the hash-bang `#!` command in Linux.

Any GSL script, no matter how simple, works with an XML document loaded into GSL's memory as a data tree. In our first `interest.gsl` script, the data tree contains just this:

```
<root script = "interest" />
```

GSL automatically creates this data tree when we ask it to execute a GSL script.

If, on the other hand, we ask GSL to execute an XML file, it loads this XML file into its data tree. Assuming we also asked for it, it will then execute a GSL script against that XML tree. Technically speaking, GSL searches the root item - which can have any name - for an attribute called "script". We can put attributes into the root item in several ways. One is to simply add them to the XML file, as we did. The other is to place them on the command line, like this:

```
gsl -script:interest deposits.xml
```

All variables that we define and use are stored in the data tree, somewhere. This is the only data structure that GSL scripts work with, and it can get very complex. For many people, understanding this complexity is the most difficult thing about using GSL - hierarchies of data are one of those things most human brains do not handle very well. We use abstractions like XNF to make this simpler, but that is something I'll discuss later.

2.2 Templates and Scripts

GSL uses the term "template" to describe text that is output as generated code. GSL works in two modes - script mode, and template mode. When you execute a GSL script directly, as we did in the first example, GSL starts in script mode. When you execute a GSL script indirectly, through an XML file, as we did in the second example, GSL starts in template mode. Try removing the `.template 0` and `.endtemplate` lines and you'll see what I mean. The script just gets copied to the output stream, the console, by default.

In template mode, GSL commands start with a dot in the first column. In script mode, all lines are assumed to be GSL commands unless they start with `>` (output) in the first column, in which case they are handled as template lines.

Script mode is useful when you are doing a lot of GSL scripting work. Often you need to prepare data, check the XML tree, and so on, before you can start to generate code. Template mode is useful when you want to output a lot of data, or actually want to generate code.

You can mix GSL commands and template code by putting a dot at the start of lines with GSL commands. Like this:

```
.while year < years
.   accumulated = accumulated * (rate / 100 + 1)
.   year = year + 1
.endwhile
```

I'm now going to generate a little HTML report of the different calculations. The listing below shows the third version of `interest.gsl`:

```
.output "deposits.html"
<html>
<head>
<title>So You Want To Be A Millionaire?</title>
</head>
<body>
<h1>So You Want To Be A Millionaire?</h1>
<table>
<tr><th>Original amount</th>
<th>Interest rate</th>
<th>Term, years</th>
<th>Final amount</th>
</tr>
.for deposit
.   year = 1
.   accumulated = amount
.   while year < years
.       accumulated = accumulated * (rate / 100 + 1)
.       year = year + 1
.   endwhile
<tr><td>$(amount)</td>
<td>$(rate)%</td>
<td>$(years)</td>
<td>$(accumulated)</td>
</tr>
.endfor
</table>
</body>
</html>
```

Note these syntax aspects:

- `output <expression>` - Start sending output to the filename specified
- $(name) - Insert value of attribute in output text

To produce the HTML report run the same command as before:

```
gsl deposits.xml
```

And then load deposits.html into your browser to see what it looks like.

If you're a web developer with any experience, you will see right away what's happening. We're generating a web page dynamically, just like a hundred other web tools. But there are significant differences:

Unlike a dynamic web page, here we explicitly specify the output file ourselves, using the "output" command. We can output zero, one, or a hundred different files if we want to.

We're working off a data tree that can be as complex as we want. Each "for" loop opens a new scope, acting on a set of child entities. A dynamic web page works off some flat data, coming from the browser or a database. You can make web pages that work on a hierarchical data set, but it's extra work.

GSL lets you load and navigate XML data so easily that you don't even realize you're busy. The combination of an explicit script language like GSL plus a hierarchical XML data tree works well.

2.3 Modeling a Web Site

I'm going to propose a simple abstract model for a web site, as an example. When you understand this example, you'll have a much better idea of how we design new models, so that you can design your own.

To start with, I'll explain how I design a new model, and then I'll take you through the steps of building a code generator that brings it to life.

Our model lets us build simple web sites. A web site is a mixture of different types of document, for instance:

- HTML pages for the content.
- JavaScript for menus.
- CSS style sheets for look and feel.
- Images for icons and for cosmetics.

And so on. When we make a new model, it's worth asking the question, "how would I make a thousand of these?" I.E., a thousand web sites. Well, we'd have lots of content, which would be different for each web site, possibly with some common parts. The content could definitely be based on standard templates - it's unlikely we'd make each of a thousand sites entirely from scratch.

If we used JavaScript menus, we'd presumably use the same code in each site, changing only the menu content to match the structure of the site.

Most likely we'd use a unique CSS stylesheet for each site, to give each site a unique look and feel, but they could also be based on a standard template.

Finally, the images and icons would be a mixture of standard graphics and customised graphics, depending on how pretty we want each site to look.

Our model is going to be the basis for code generation, that is, the mass production of as much of the above as is reasonable. To do this, we need to make a compact and efficient statement of exactly what is needed to produce each web site.

It's like constructing a thousand houses. It's expensive to design and build each house as a unique thing. It's much cheaper to make a single common plan, and then for each house, state the differences. So one house might have a different roof shape, while another has larger windows, but all houses share the same materials, wall and floor construction, and so on.

When we mass produce something, we're clearly aiming for low cost and consistent , and hopefully high, quality. It's the same with code generation. So, let's get to our web site model. What information do we actually need to specify?

- First, we need to know all the pages in the web site, so that we can build menus.
- Second, we need basic information for each page. Typically, I like to define a title and subtitle, an image (for pretty marketing purposes), and a block of content (which can be raw HTML).
- Third, we some information for all pages - for example, a logo and a copyright statement.

The next step is to sketch a model that can hold this information in a useful way. Remember that we use XML as a modeling language. So, we invent an XML syntax for our model. For each page, I'd like to write something like this:

```
<page
    name = "name of page"
    title = "Title text goes here"
    subtitle = "Subtitle text goes here"
    >
<content>
Content HTML goes here
</content>
</page>
```

When I design new XML languages like the above, I use entity attributes to hold single-line properties, and child entities to hold multi-line properties or properties that can occur more than once. It just seems more elegant than putting properties in child entities, since this implies those properties can occur many times. It does not make sense for a page to have more than one name, title, subtitle, or image in our model, so we define these as attributes of the page entity. The iMatix MOP tools use this style very heavily.

Once we've defined a set of pages, how do we tie these together into a web site? Let's use a second model for the overall web site:

```
<site copyright = "copyright statement goes here">
<section name = "name of section">
    <page name = "name of page" /> ...
</section>...
</site>
```

I've defined a `<section>` tag that breaks the pages into groups. Now let's jump right in and make ourselves a web site. There's no better way to test a model than to try using it. As an example, I'll make a new web site for my local grocer, who has decided, finally, to go on-line.

2.4 First Draft

We'll make the web site as several XML files. This is a design choice. We could also make the site as a single large XML file. It's a trade-off between ease of use (a single file is easier in smaller cases) and scalability (it's not practical to edit a large site with hundreds of pages as a single file).

To start with, we'll define the overall site like this:

```
<?xml version = "1.0" ?>
<site
    copyright = "Copyright &#169; Local Grocer"
    script = "sitegen_1.gsl"
    >
<section name = "Welcome">
    <page name = "index" />
</section>
<section name = "Products">
```

```
    <page name = "fruit" />
    <page name = "vegetables" />
</section>
</site>
```

Note the first line, which defines the file as XML, and the `script` tag, which tells GSL what script to run to process the data. We've defined three pages. Let's write very a simple version of each of these:

Next, we will write three more short XML files as shown below. First the index page:

```
<page
    name = "index"
    title = "Local Grocer"
    subtitle = "Visit the Local Grocer"
    >
<content>
<h3>Close to you</h3>
<p>We're just around the corner, if you live near by.</p>
<h3>Always open</h3>
<p>And if we're closed, just come back tomorrow.</p>
<h3>Cheap and convenient</h3>
<p>Much cheaper and easier than growing your own vegetables and fruit.</p>
</content>
</page>
```

Next, the fruit page:

```
<page
    name = "fruit"
    title = "Our Fruit Stand"
    subtitle = "Lucious Tropical Fruits"
    >
<content>
<h3>Always fresh</h3>
<p>Just like it was plucked from the tree last month.</p>
<h3>Special deal</h3>
<p>Any five pieces of fruit, for the price of ten!</p>
<h3>Money back if not satisfied</h3>
<p>We'll give you your money back if we're not satisfied with it!</p>
</content>
</page>
```

and last the vegetable page:

```
<page
    name = "vegetables"
    title = "Our Vegetables"
    subtitle = "Healthy Organic Vegetables"
>
<content>
<h3>100% organic vegetables</h3>
<p>All vegetables made from cardon, oxygen, and hydrogen molecules
with trace elements.</p>
<h3>Country fresh style</h3>
<p>We don't know what that means, but it sounded nice!</p>
```

```
<h3>Unique take-away concept</h3>
<p>Now you can consume your vegetables in the comfort of your own home.</p>
</content>
</page>
```

Finally, here is the first draft of the web generation script. It does not produce anything, it simply loads the web site data into an XML tree and then saves this (in a file called `root.xml`) that we can look at to see what live data the script is actually working with:

```
.###  Since we run the script off the XML file, it starts in
.###  template mode.
.template 0
 for section
    for page
        ###  Load XML <page> data
        xml to section from "$(page.name).xml"
        ###  Delete old <page> tag
        delete page
    endfor
 endfor
 save root
.endtemplate
```

Let's look at what this script does. First, it switches off template mode so we can write ordinary GSL without starting each line with a dot. GSL starts scripts in template mode if they are launched from the XML file. It's useful in many cases but not here. So, we wrap the whole script in `.template 0` and `.endtemplate`.

Second, the script works through each section and page, and loads the XML data for that page. It does this using two commands, `xml` and `delete`. The first loads XML data from a file into the specified scope (`<section>`, in this case), and the second deletes the current page (since the loaded data also contains a `<page>` tag).

Finally, the script saves the whole XML tree to a file. If you want to try the next steps you must have installed GSL, as I described in the last article. Run the script like this:

```
gsl site
```

GSL looks for the file called `site.xml`. When the script has run, take a look at `root.xml`. This shows you what we're going to work with to generate the real HTML.

2.5 Inserting Variables

When we generate output, we insert variable values into the generated text. This is very much like using shell variables.

GSL does automatic case conversion on output variable. This is very useful when we generate programming languages. For example, the $(name) form outputs a variable in lower case:

```
output "$(filename).c"
```

The $(NAME) form outputs the same value in uppercase:

```
#if defined ($(FILENAME)_INCLUDED)
```

And the $(Name) form outputs the variable in title case, i.e. the first letter is capitalised:

```
##################  $(Filename)   #################
```

One side-effect of automatic case conversion is that we'll often get variables converted to lower case simply because we used the $(name) form. If we don't want a variable to be automatically case converted, we use this form: $(name:). This is also called the 'empty modifier'.

A second side-effect of automatic case conversion is that variable names are not case sensitive. By default GSL ignores the case of variable names so that $(me) and $(ME) refer to the same variable.

But putting empty modifiers in every variable expansion gets tiresome, and GSL lets us switch off automatic case conversion, using this instruction:

```
ignorecase = 0
```

This tells GSL, "variable names are case sensitive, and do not convert variable values on output".

2.6 Looping through Trees

In our first draft we loaded each page into the XML tree and deleted the original page definition. That was this text:

```
for section
    for page
        xml to section from "$(page.name).xml"
        delete page
    endfor
endfor
```

To generate output for each page, we're going to iterate through the sections one more time. Since we're deleting old `<page>` entities and loading new ones from the XML definitions, we need to iterate through the sections and pages over again. This is the code that generates the output for each page:

```
for section
    for page
        include "template.gsl"
    endfor
endfor
```

The include command executes GSL code in another file. We're going to do all the hard work in a separate file, which I've called `template.gsl`, so that it's easy to change the HTML generation independently from the top-level GSL code. This is good practice for several reasons:

It's nice, in larger projects, that each big code generation task sits in its own file where it can be owned by a single person.

We can add more templates - to produce other types of output - for the same model very easily and safely.

And you'll see in later examples that we tend to write a single GSL file for each output we want to produce. In XNF - the tool we use for larger-scale code generation projects - these scripts are called "targets".

2.7 Building the Output

The HTML template looks like this:

```
.template 1
.echo "Generating $(page.name) page..."
.output "$(page.name).html"
<!DOCTYPE...>
<html>
   ...
</html>
.endtemplate
```

Most of it is fairly straight-forward, though you do need to understand how XHTML and CSS work (and I'm not going to explain that here).

- The echo command tells the user what's going on. It's polite to do this, although in realistic cases we'll also let the user suppress such reports using a 'quiet' option.
- The output command creates the HTML page.
- The text `<!DOCTYPE...>` to `</html>` is the body of the page, which I'll explain below.

The template starts by setting template mode on. This means that any GSL commands we want to use here must start with a dot. It makes the HTML easy to read and to maintain.

Let's look at the chunk of code that produces the site index. This is - in our version of the web site generator - a menu that is embedded into each page. The CSS stylesheet can place this menu anywhere on the page. Here is the GSL code that generates it:

```
.for site.section
<h3 class="menu_heading">$(section.name)</h3>
<ul class="menu_item">
.   for page
<li><a class="menu_item"
    href="$(page.name).html">$(page.title)</a></li>
.   endfor
</ul>
.endfor
```

The interesting thing here is that we say for site.section in order to iterate through the sections. The site. prefix is a parent scope name, it tells GSL "look for all sections in the current site". If we don't use the scope name, GSL would look for all sections in the current scope (the page) and find nothing. This is a common beginner's error.

Note that the parent scope is not always needed. These two blocks do exactly the same thing:

```
.for site.section
.   for page
.   endfor
.endfor
```

and:

```
.for site.section
.   for section.page
.   endfor
.endfor
```

But the first form is simpler and I recommend you drop explicit parent scope names when you are "tunneling into" the XML data tree.

Near the end of the template you see this construction:

```
.for content
$(content.string ())
.endfor
```

What is going on here? The answer is, we're grabbing the whole `<content>` block, including all the XML it contains, as a single string. Conveniently, XHTML is also XML, so we can read the XHTML content block as part of our XML data file. As a bonus, GSL will also validate it and tell you if there are errors, such as missing or malformed tags.

The string() function returns a string that holds the XML value of the specified entity. For the index page, it returns this value (as a single string):

```
<content><h3>Close to you</h3><p>We're just around the corner, if you live near by.</p><h3>Always open
```

When we enclose this in `$(` and `)`, it writes the string to the current output file. Thus we generate the body of the web page.

2.8 Putting it All Together

In our first draft we read the XML data from several files and we constructed a single tree with all the data we needed to generate code. This two-pass approach is the way I recommend you construct all GSL code generators:

- First, load all data into a single memory tree, denormalise and validate.
- Second, generate code from that single memory tree.

The final web site generator consists of three pieces. Here is the revised web site generator.

```
.### Since we run the script off the XML file, it starts in
.### template mode.
.template 0
 ignorecase = 0
 for section
    for page
        xml to section from "$(page.name).xml"
        delete page
    endfor
 endfor
 for section
    for page
        include "template.gsl"
    endfor
 endfor
.endtemplate
```

Here is the template for the HTML output.

```
.###  This whole script runs in template mode.
.#
.template 1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <title>$(page.title)</title>
  <link rel="stylesheet" href="default.css" type="text/css"/>
</head>
<body>
  <div id="left_container">
    <div id="logo_container">
      <a href="index.html"><img id="logo" src="$(page.name).jpg"/></a>
    </div>
    <div id="menu_container">
.for site.section
      <h3 class="menu_heading">$(section.name)</h3>
      <ul class="menu_item">
.   for page
```

```
        <li><a class="menu_item" href="$(page.name).html">$(page.title)</a></li>
.   endfor
      </ul>
.endfor
      <h3 class="menu_heading">Copyright</h3>
    </div>
    <div id="copyright">
      <p>$(copyright)</p>
    </div>
    <h3 class="menu_heading"> </h3>
  </div>
  <div id="right_container">
    <div id="title_container">
      <h1 id="title">$(page.title)</h1>
      <h2 id="title">$(page.subtitle)</h2>
    </div>
    <div id="content_container">
    <!-- Page content -->
.for content
    $(content.string ())
.endfor
    <!-- End page content -->
    </div>
  </div>
</body>
</html>
.endtemplate
```

To build the final web site, make sure the `site.xml` specifies the correct script:

```
<site
    copyright = "Copyright &#169; Local Grocer"
    script = "sitegen.gsl"
    >
```

And then build the web site using the same command as previously:

```
gsl site
```

The HTML template and the CSS file are made for each other. Note that:

- The HTML template assumes that each page has an image file with the name of the page, and extension "jpg".
- The colors and layout of the pages is defined in the CSS stylesheet.
- The menu is generated into each page.

2.9 Exercise for the Reader

It's an interesting exercise to re-implement our code generator using other code generation tools. For example, if you're familiar with XSLT, try building the web site generator using that. You may find you need to cheat, for example putting the whole web site model into a single file.

2.10 Extending the Model

I've shown you how to design a simple model, and bring it to life using GSL. This web site generator is actually based on one that I use for some of my own web sites. You can extend this model in many directions, for instance:

- You can change the type of menu, using a JavaScript drop-down menu instead of static HTML links.
- You can define your own modeling language for the HTML content.
- You can add other concepts and idioms to the model, depending on what you need in your web site.

But most of all, the point of this example is to teach you how to use GSL in your daily work. As you've seen, it's easy to create models, and it's easy to change them. This is the secret of code generation - you don't need to get it right the first time. Models are hard to get right. So go ahead and experiment, since GSL makes it cheap to change your mind.

3 Model-Oriented Programming

This article is aimed at the professional programmer. I'm going to attack a complex subject, something that few people know about. It's a new way of programming called "model-oriented programming". I'm not going to ask you to throw out your programming languages or tools. MOP works as a layer on top of everything you know today. I am going to ask you to rethink what it means to "write a program", and to see that most of the code you write could be better written by robots, meaning other programs. And I'm going to teach you how to design and make such robots.

MOP works for every kind of area you write code for. Whether you write games, Linux drivers, servers, applications, plug-ins, whether you use Java, C, Perl, Ruby, Python, Gnome or KDE... once you start to see the world as models you'll find yourself writing more code, faster, than you ever thought possible.

In this article you will learn what MOP is, and why we invented it. I'll also explain some of the underlying technologies.

Be warned. This might hurt a little. All I can promise is that if you learn to use MOP you will use for the rest of your life, and wonder how you ever worked without it.

3.1 Becoming a Very Good Programmer

I've learned a few things about software since I wrote my first small program in 1981 or so. First: if it's not impossible, it's not worth doing. Second: software design is about overcoming human limitations, not technical ones. Third: very few people can actually design good, useful, large-scale software systems, which for me is the goal of programming.

In my experience there are these four main aspects to becoming a very good programmer (which I hope to become, one day):

- Never throw out anything that works until it is really worn out. This mainly means writing portable code.
- Never solve the same problem more than once in parallel. This mainly means building tools.
- Solve the same problem often in serial. This means being willing to throw out code and rewrite it when you find better ways.
- Write code, write code, write code, until it is as natural as speaking.

Of course you also need talent, opportunity, and guidance, but a focus on portability, obsessive tool building, and years of practice can turn talent into real skill.

3.2 Tools that Write Software

In this article I'll focus on the second part, making tools. There are quite a few metaphors for software tools. For example, the Unix metaphor consists of tools as filters: read some data, do some work, produce some output. It's a simple model that lets you chain together tools. Linux has many tools that work as filters.

A more subtle but much more powerful metaphor is to build tools as languages. That is, when you come across a new class of problem, you create a new language that lets you express solutions to those problems in a simple and clear way.

Programming languages are one example of tools that work like this. Most programming languages have their strong and weak points, but basically they are all equivalent: they solve the general problem of "programming", not specific problems like "constructing a firewall" or "building a static web site".

Now consider HTML. This is a language that takes a different approach. You don't use HTML to write programs: you use it to define structured documents, and then you give these documents to programs that can do useful things with the definitions, like show them on a screen.

It's worth comparing HTML to a language like PostScript, also used to get documents looking pretty on paper or screen. PostScript is a programming language (a threaded stack-based interpreter descended from Forth, in fact). People have written, for example, web servers in PostScript. No writer or designer actually sits down and writes PostScript, though people did this before HTML existed.

I sometimes use a tool that turns HTML pages into PostScript documents. Now, as a writer, I can use HTML to write my documents and then push a button to turn this HTML into PostScript. What I am actually doing is converting a descriptive language into a programming language. A HTML-to-PostScript converter lets millions of non-technical people suddenly produce perfect programs at will. Millions of people who never think of themselves as "programmers" can write PostScript, via higher-level abstractions like HTML. And the PostScript programs they produce are much better than an average PostScript programmer can write in a reasonable time.

3.3 Abstractions and Modeling Languages

HTML is in fact a "modeling language", a language used to describe some system or entity. Modeling languages are very interesting because they provide levels of abstraction that programming languages cannot even conceive of. Abstraction lets you define and use high-level concepts like, "this is a document title", without having to specify what that actually means, on paper or on the screen. Abstraction relies on concepts that have enough meaning to be useful, without being too detailed. For example, "the web" is a useful abstraction for "various URL formats, protocols for transferring such resources, clients and servers that implement such protocols, and collections of resources that are thus interconnected".

Abstraction is an important concept. It is one of the keys to solving complex problems. Let me give you another example of abstraction. We can build an application using a shell script that does each step. This is not abstract, it is a literal set of steps. A makefile is more abstract: it adds the abstractions of "file type" (based on extension) and "target" and specifies how we transform one type of file into another in order to build a specific target. There are more abstract descriptions of projects too. Each time we make a good abstraction - a simple view that hides complex details - we eliminate a layer of manual work.

Now imagine you could use modeling languages as a way of writing programs. Instead of writing Perl, or Java, or C++, or Ruby, you would describe some kind of model and then press a button. In fact this is not a new idea: I've worked with systems that have done this, more or less successfully, for twenty years.

Historically, computer scientists have tried to make programming languages more powerful by adding functionality and by adding programming abstractions. The underlying assumption is that since programming languages are general purpose, they must be the best tools for building software. Adding general purpose programming capacity gets more and more complex as we reach for more abstraction. Thus we get languages that are so complex that to master them is a full career.

This is, I believe, a mistaken approach. Complexity is difficult to control, and complex languages (like Perl and C++) have a reputation for producing complex and hard-to-maintain code. As a programmer, I have quite a problem investing so much in any single language.

The trick that I've found (since I'm lazy and don't have the patience to read 500-page guides to programming languages) is to create simple abstractions - models - that solve useful problems, and to use these models to generate code, just as we produce documents on screen and on paper by generating code from abstractions like HTML.

For naive users, a model is a visual thing, but for us programmers, a textual modeling language is much more useful. There exist many modeling languages, and as I said, it is not a new concept. For example, in 1991 I wrote a tool, Libero, that turns finite state machine models into code. Libero was extremely useful, and it is still part of our toolbox today. What it does is take a state machine model (a text file), and turn that into code in arbitrary languages (we made code generators for twenty or so programming languages). State machines are a very useful model for writing programs, but that's a different story.

Libero took me about three months to build, time that I've won back on many projects. In a general sense, Libero is like the HTML-to-PostScript convertor. It takes a definition written in an abstract modeling language and turns that into code that makes the model come to life. The generated code is always perfect, and as invisible as the PostScript code.

3.4 Leverage to Move Mountains

I'm coming to the point of this article. This kind of model-to-code tool is very useful. It gives you leverage. That is, the ability to move mountains. It is much easier, faster, cheaper to change an abstract model than it is to change the code that makes it work. Look again at a PostScript program and now imagine the work needed to change a H1 item to H2.

Good models let you work 10 to 100 times faster than any programming language. As long as you stick to the problems the model was meant for, there is really no downside, no reason to prefer PostScript over HTML for writing texts.

I've said that part of being a world-class programmer is the ability to make useful tools. But how do you, as a programmer, make a tool that compiles a modeling language into code? You need to solve four main problems:

- You need to learn, borrow, adapt, or invent useful models. This is not easy. Good models like finite state machines and hierarchical documents took many clever people many years to invent and refine.
- You need to define a language that lets people make such models. Again, getting this right is delicate. There are hundreds of ways to write state machines, just as there were hundreds of document markup languages before HTML, most being far too complex and thus useless.
- You need to build a parser that can read this language, check it, turn it into internal structures, optimise those structures, etc. Needless to say, writing parsers is not easy, there exist whole sets of tools just to solve this problem.
- You need to build a code generator that can take these internal structures and spit out the final code in whatever target languages you want to produce. Writing code generators is a black art. There are almost no books on the subject, no standard technologies.

If you've ever studied how compilers work, it's much the same problem. What I'm talking about is building compilers for modeling languages.

3.5 Case Study - OpenAMQ

Modeling languages and programming languages can overlap. For example, objects are a type of model. The biggest problem with putting models into the programming language is that for real, large problems, we need many different types of model, and these cannot be expressed in a single language. Languages that attempt this become too complex to work with. Imagine attempting to describe a hierarchical document using objects, and compare this to writing some HTML by hand.

I'll explain with a large case taken from a real project, OpenAMQ. This is an AMQP messaging server. We used C as the target language for portability and performance, but we actually designed the software as lots of high-level models. Each modeling language was part of a code generation process that produced real code. We used modeling languages for:

- Classes to encapsulating functions.
- Finite state machines for building protocol handlers.

- Project definitions for building and packaging sources.
- Grammar definitions for building parsers and code generators.
- Grammar definitions for communication protocols.
- Test scripting language.

The key to making these different modeling tools was cost. If it was cheap to design, test, and use modeling languages, we could cut the research and learning time dramatically.

Since I wrote Libero almost twenty years ago, I'd been working with Jonathan Schultz to make better technologies for building modeling languages and the code generators that bring them to life. We finished the last of the main tools in 2005, and we then had technology that let us design and deploy new modeling languages in a matter of weeks. The process was so fast, and so efficient, that we were able to generate almost 100% of the middleware server, a half-million lines of C code, from about sixty thousand lines of modeling code.

The downside is that anyone wanting to understand the code had to learn the five or six models we use. The upside is that they only had sixty thousand lines of code to read, not half a million.

Let me take you through the main parts of our architecture:

- The basic technology is GSL, the programming language that we use to build code generators. Yes, you can write a web server in GSL, but that would be pointless. GSL is aimed very much at spitting out huge volumes of perfect code. GSL is an interpreter, it runs as a command, just like Perl or another scripting language.
- The second main technology is XML, which we did not invent of course, but which we happily adopted in 1997, having spent several years designing our own very similar meta-languages. We use XML in a simplistic way, to model data, not to do any kind of complex document manipulation. No stylesheets or namespaces, thus.
- The third main technology is XNF, which is a tool for building model-driven code generators. We start to get meta here. XNF (for "XML Normal Form") lets you define the grammar of an XML-based modeling language. From that grammar XNF produces parsers and a framework into which you plug hand-written back-end code generators. XNF is a modeling language for code generators. XNF is the basis for all our complex modeling tools, including XNF itself.

These tools - which are included in the OpenAMQ distribution's `base2` project - are somewhat unusual. The techniques of code generation are not well understood, and no teams have ever pushed these techniques as far as we have. I don't promise that it will be easy to understand - abstraction can be hard to grasp - but once you "get it", you'll be able to produce tools that solve your programming problems ten times faster than using any other technique.

3.6 Other Model-Driven Architectures

Using models as the basis for designing applications is not new. I've worked with many tools that promised "an end to programming" through the magic of point and click modeling. Some of these - such as UML (Universal Modeling Language) - have become industry standards. In my experience, these tools do not work except as expensive and slow documentation tools.

Perhaps my opinion of classic modeling tools such as UML have been influenced by watching them being abused on large projects. The typical scenario is that a big team of analysts work for a year to produce a "model", which is then thrown out as a second team of developers write the actual code.

The fundamental problem is that no single modeling language can cover the variety needed to solve real world programming challenges. Just imagine someone suggesting that UML could be used to write a Linux device driver, or a high-performance game. That's a joke! Yet my team uses models to design and build very technical, very high-performance software. You just need the right models.

Classic MDA tools attempt to do everything with a single modeling language. This is doomed to failure except within a very narrow niche of work. Indeed, it is more expensive to "not write code" using a language like UML

than to simply write the code in Java. Just as a single programming language cannot cover all abstractions, neither can a single modeling language.

To succeed with a model-driven architecture, you need a way to build, test, and improve a variety of different models, each solving one specific domain. What you need is not a single, do-it-all modeling language, but a technology that lets you build arbitrary modeling languages.

3.7 Why use MOP?

Despite the trendy name, MOP is really about solving real problems in the most efficient possible way. Let's look at the main advantages my team gets from using MOP:

- We have to write much less code to get the same results. I call this "leverage". One line of modeling code can be worth ten or twenty lines of a programming language. Using less code has many knock-on advantages: we work faster, better, and cheaper.
- We get high-level models of important aspects of the system. All systems have key models, but they are usually hidden in the code and impossible to verify, formalise, or exploit fully. When the model is turned into a concrete language, it makes the software much better.
- We can produce extremely high-quality code. This is an effect of doing code generation: the generated code we produce has no errors, and is as good as a human programmer can write, consistently.
- We write less internal documentation, and often none at all, since each model is documentation.
- We are immune to technological changes since MOP is entirely abstract from specific programming languages, operating systems, and trends. It can take years to develop really good models but they work for decades.

There are also disadvantages:

- People do not rapidly understand or trust the approach. I've been accused of over-investing in tools (sometimes more than half the cost of a project goes into modeling tools). The look on the client's face when we deliver version after version of impeccable software in impossibly short deadlines is worth it.
- Programmers do not rapidly understand the models. It takes time to learn each one, sometimes weeks or months.

So, MOP is best used in small, skilled, and long-lasting teams (like iMatix) that solve highly complex and critical problems. Before you can use MOP in a project you need complete confidence of the people paying for the work. If you're writing software for yourself, it's easy. If you're writing software for other people, this can be a hard sell. MOP can also be used to give structure to larger development teams, but it is a lot of work to train mediocre people to use sophisticated models.

The sad thing in the software business is that few people actually understand that better techniques save money. Still, there is no pleasure, as a programmer, in writing bad code using bad tools. So, learn to use MOP, then convince your bosses that they will save money, right away, by using this. Everyone wins.

3.8 A Short History of Code Generation

To understand and use MOP you need to appreciate code generation as a technology. I first started writing code generators in 1985, and I've seen these tools evolve through several stages (in my own work, but also in the general domain):

- *Hard-coded code generators* that take some meta-data (a model) and output code using print statements. This is the most common, and the most limited form. Typical examples are all the classic "code generators" built into products.
- *Template-driven code generators* that use symbolic insertion to inject meta- data into a template. This technology makes it much easier to write new ad-hoc templates. Typical examples are any technology that produces dynamic web page.

- *Scripted code generators* that use a high-level language to manipulate meta- data and then inject it into templates. This technology makes it much easier to write new ad-hoc code generators. Typical examples are XSLT, GSL and some other scripted code generation languages.

- *Meta code generators* that build code generators. This technology makes it possible to construct code generators for very complex modeling languages. The only examples we know are iMatix tools such as XNF and ASL.

Our first versions of GSL were born in 1995. We moved onto XML in 1997. The problem of writing the code generator scripting language (GSL) is largely solved, and at iMatix we don't consider this a priority task. At some point we want to rewrite GSL to be a lot faster, but what interests us now is using MOP techniques to solve difficult problems.

3.9 Myths about Code Generation

Code generators are often seen as a technological burden, rather than useful tools. I suspect that this is because primitive code generators (which covers most code generators) are so painful to use. Some of the common myths about code generation are:

- *Code generators only work for simplistic cases.* This is often true, but only because most code generators are simplistic.

- *Generated code is unreadable, and generated code is low quality.* This is often true because their authors focus on the application-specific problem, rather than on making the template easy to modify and improve. In a template-based code generator the code can be as good as or better than hand-written code.

- *Code generators are expensive to make.* This is typical of hard-coded code generators where the slightest change to the template means modifying, compiling, linking, and distributing a new release of the code generator.

- *Code generators are too much effort to use.* This says more about tool designers than about the problems that the tools solve.

- *Code generators are complex.* This is true: all abstractions are internally complex - look at the work required to write a good web browser or web server.

GSL solves most of these problems, and even a beginner can make useful code generators for interfaces, database management, XNF solves the last problem - it applies MOP to MOP itself, letting us make sophisticated code generators much more rapidly than by hand.

3.10 The Correctness of Generated Code

When you use a tool that produces large amounts of code for you, you will naturally ask, "how do I know the code is correct". You need to be able to trust your tools.

A code generator, luckily, is not random. It is like a simple compiler: take high-level construct, translate into target code. If there is a bug in this translation step, you will get target code that has bugs. Luckily, it's easier, not harder, to get correct code from a machine than by hand. I'll explain how we do this:

- Overall, we are quite strict about how we build our code generators. That makes bugs in the code generators rarer, and easier to find.

- When we start a new code generator, we build it gradually, and of course we inspect the code that it produces as we develop it.

- As we make the code generator more sophisticated we build a regression test suite that lets us catch any new errors in old code very rapidly.

In any programming environment, the key to producing good code is to test heavily, and to use appropriate automation, i.e. tools. When we say the "correctness of generated code", we really mean, "how to avoid bugs in the code generator".

Since we use the MOP approach to build the code generators themselves, we get very good code generators, cheaply. It is the same concept as a "self-hosting" compiler. On many projects where we've used MOP, I'm able to deliver hundreds of thousands of lines of code, and say, with confidence, "there is not a single bug in this code".

4 GSL/4.1 Reference Manual

4.1 Command-line Syntax

To run GSL, use one of the following syntaxes:

```
gsl -<option> ... -<attr>[:<value>] ... <filename> ...
gsl -a -<option> ... -<attr>[:<value>] <filename> <arg> ...
```

If the filename has no extension, GSL tries to find an XML file with that name, or with the extension `.xml` (recognised by the `<?xml...>` tag on the first line). If it finds no XML file it tries to find a file with that name or the extension `.gsl`, which it interprets as a GSL file.

Options currently recognised by GSL are:

- -a argument: Pass arguments following filename to GSL script
- -q quiet: suppress routine messages
- -p parallel: process files in parallel
- -s:n size:n set script cache size - default is 1000000
- -h help: show command-line summary
- -v version: show full version information

Command-line attributes are loaded with an XML file and are available to a script. This allows paramaters to be passed from the command line to the script. The attribute script can be set to the name of a GSL file to be interpreted.

If GSL found an XML file, it loads it, then looks for an attribute named script of the top-level item. This name is used to find a GSL script to interpret. If GSL found a GSL file, it begins interpreting it without loading an XML file.

4.2 Concepts

4.2.1 Scalar Data Types

GSL recognises two scalar data types: numeric and string. It generally makes no formal distinction between them; if a value looks numeric, then it is treated as such, otherwise it is treated as a string. If strict typing is required, the type conversion functions `conv.number` and `conv.string` can be used.

4.2.2 Structured Data Types

GSL also understands structured data types. Structured data types are modelled on XML; they have a name, attributes and children. Attributes and children may be of a scalar type or (unlike XML) a structured type. An attribute effectively represents a 1:1 link while children represent 1:n links.

Structured data types are used to represent underlying data, and the attributes and children are based on their structure. The archetypal case is XML data; its particularities mean that children are also XML items, attributes may only have scalar values while both children and attributes may have any name.

4.2.3 Constants

Constants express a constant value of one of the two scalar types. A string constant is specified with either single- or double-quotes as delimiters, for example: `"ABC"`. String constants may continue over several source lines. The line break is considered part of the string constant, unless the last character in the line is a single backslash (\) in which case neither the backslash nor the line break is part of the string. A numeric constant is a simple number with an optional sign and optional decimal characters, for example 123 and -0.3.

4.2.4 Scopes

According to The Free On-line Dictionary of Computing, © 1993-2004 Denis Howe:

> The scope of an identifier is the region of a program source within which it represents a certain thing. This usually extends from the place where it is declared to the end of the smallest enclosing block (begin/end or procedure/function body). An inner block may contain a redeclaration of the same identifier in which case the scope of the outer declaration does not include (is "shadowed" or "occluded" by) the scope of the inner.

GSL extends this usage so that a scope also has an alias, or name. When we refer to a scope, we generally do so by its alias rather than by the region in the script where it is defined.

All data access in GSL begins with a scope; that is, there is no such thing as data unrelated to a GSL scope. Note that it is not always necessary to explicitly specify the scope (see 4.2.4 below).

A scope defines a mapping from GSL data space onto the underlying data structures. Its name is typically the same as the name of its underlying structure; the principal reason for using a different alias is to differentiate two data structures with the same name.

Scopes are opened (created) and closed (terminated) by matching pairs of GSL instructions: for/endfor, new/endnew and scope/endscope. These instructions must be nested; that is they may not overlap. At any point in the script there is therefore a series of open scopes, ranging from outer (opened earlier) to inner (opened more recently).

In addition, GSL predefines a number of scopes; these are considered to have been opened before any scopes opened by the script.

Referencing Scopes In general, scopes may be referred to in a variety of ways: by alias, by number or implicitly. When referring to a scope by its alias name, open scopes are searched, from innermost to outermost, for a scope with the specified name. If there is more than one scope with the same name, only the innermost can be referred to by name. In addition, scopes may be declared with no alias, in which case they cannot be referred to by name.

When referring to a scope by number, 1 refers to the outermost scope, 2 to the second outermost scope and so on. Alternatively 0 refers to the innermost scope (this is very useful), -1 refers to the second innermost scope and so on. In fact the situation is slightly more complicated than this. Scopes may be declared as 'unstacked', meaning that they do not appear in the sequence of numbers, either positive or negative.

A unstacked scope with no alias could not be referred to at all and is thus disallowed.

A scope may be recast with a new alias, stacked or unstacked, with the `.scope` instruction.

Predefined Scopes Before processing a script, GSL defines several scopes. These need to be taken into account when referring to a scope with a positive number.

- The first predefined scope is called `global` and can be used to hold global data. Its underlying data structure is a symbol table; it may have attributes with any name and any type (scalar or structured), but no children.
- The second predefined scope is called `gsl` and holds GSL internal data. NB. At the current moment there is a namespace clash between the scope `gsl` and the instruction `gsl`. This can be avoided by the use of square brackets [gsl] to reference the scope. The problem will disappear when the gsl instruction is deprecated.
- The third predefined scope is called `class` and holds the classes registered with GSL. These are typically: file, string, env, conv, math, sock, thread, xml, time, regexp, proc and diag. They should be documented below.

- The fourth predefined scope is called `root`. It refers to an XML item also called `root`, which is typically used as a parent for all other XML structures.
- The fifth and final predefined scope is the top-level XML item from the XML source file, if one was specified.

4.2.5 Data Specifiers

A data specifier is the means by which you access an item of data. GSL provides a variety of ways to access a particular piece of data.

Scopes As mentioned above, all data begins with a scope. The underlying data of a scope may be accessed by simply referencing the scope. For example

```
global.foo = root
```

assigns the XML structure referred to by the scope `root` to the attribute `foo` of the scope `global`. Recall that the underlying data of this scope is a symbol table whose attributes can hold structured data.

Attributes Attributes are referenced by the use of the period (`.`) For instance to display the value of the attribute `name` of the XML structure referred to by the scope `root` you could use:

```
echo root.name
```

Implicit Scope Referencing If, instead of explicitly specifying the scope `root` in the above example, you used:

```
echo name
```

GSL would search stacked scopes, from the innermost to the outermost, for one that defines the attribute `name`. Assuming the scope `root` does not define an attribute `name` but the scope `global` does (we understand that neither the scope `gsl` nor the scope `class` defines an attribute `name`), this code would output the value of the attribute `name` of the scope `global`.

This form of GSL is useful for two reasons. Firstly it makes for shorter and easier-to-read code, when the location of the attribute is not in question. Secondly it allows the value to be inherited from outer to inner scopes.

Notice that the above example contains some ambiguity: does `name` refer to an attribute `name` or a scope `name`? GSL searches first scopes then attributes within scopes to find a match. If you wish to match only an attribute, then use the alternative form:

```
echo .name
```

Structure Flattening What if, in the above example, the scope `global` defined an attribute `name`, but this attribute were not a scalar value but an XML structure. Since the instruction `echo` can only deal with scalar values, it requests a flattened value of the structure. The meaning of a flattened value depends on the structure in question; for an XML item it refers to the value of the item.

Look at the example:

```
global.name = xml.load_string ("<A value = \"2\">Hello</A>")
echo name
echo global.name.value

2004/09/20 16:36:25: gsl/4 M: Hello
2004/09/20 16:36:25: gsl/4 M: 2
```

In addition to this implicit flattening, you can also explicitly request the flattened value of a structure as follows:

```
global.foo = name.
```

Notice that this form resembles a request for an attribute with no name; this is the desired effect.

Navigating Children Just as the period (.) accesses an attribute of a structure, the member (->) construct accesses a child. For instance

```
global.parent->child
```

accesses the (first) child called `child` of the structure referred to by the attribute `parent` of the global scope.

A more sophisticated version of this structure exists:

```
global.parent-> child (value = "2")
```

accesses the (first) child called `child` for which the condition `value = "2"` is TRUE.

The detail of how this works is that a scope called `child` is opened for the duration of the evaluation of the condition. This underlying data for this scope is the child (called `child`) of global.parent. The typical use for such a structure is when the child defines an attrbute `value` so that the expression is used to locate the particular child we want, the one whose attribute `value` is 2.

As a final detail, the scope opened during the evaluation of this expression may clash with other scopes called `child`. For this reason, the following expression may be used:

```
global.parent-> child (baby.value = child.value, baby)
```

The second argument `baby` indicates that the name of the scope created to evaluate the condition should be `baby`.

Identifiers An identifier is the generic term used for GSL scopes, attributes and children. It is restricted to a combination of alphanumeric characters and the underscore (_) and must begin with a letter or an underscore.

As the underlying data may not have the same restrictions on its name space, and because GSL has a certain number of reserved words which may clash with identifiers, identifiers may be surrounded by square brackets. Inside square brackets, just about anything can be used. Use a backslash (\) tn introduce a special character, two backslashes to make a backslash.

Case Sensitivity GSL has two modes of handling the case of identifiers. In the default mode, GSL matches names without regard to the case (upper or lower) used to specify them. In certain substitutions GSL modifies the case of the value of the identifier to match the case used to specify the attribute name. In case-sensitive mode, GSL matches names taking into account the case, and does not modify the case of the result. See the description of subsitutions for details. To change modes, set the value of the identifier `ignorecase` in the gsl scope to 0 or 1. Eg: `[gsl].ignorecase = 0`.

4.2.6 Expressions

GSL expressions are much the same as expressions in other high-level programming languages. They include the following operators:

Multiplicative `*, /`

Additive `+, -`

If/Default `??, ?`

Comparative `=, <>, >, >=, <, <=`

Safe comparative `?=, ?<>, ?>, ?>=, ?<, ?<=`

Logical `|, &, !`

Operator precedence is standard (multiplicative, additive, if/default, comparative, logical) and brackets are treated as you would expect. Logical operators treat zero as FALSE and non-zero as TRUE.

GSL optimises expression evaluation to the extent that the second operand of a binary logical operator (`|`, `&`) is not evaluated if the result of the expression is determined by the first operand. This allows you to use expressions such as

```
defined (X) & X
```

since the second operator is not evaluated when X is undefined. The default operator allows undefined expressions to be replaced by another expression. The value of

```
<expr1> ? [<expr2>]
```

is equal to the value of `<expr1>`, if defined; otherwise it is equal to the value of `<expr2>`, whether or not the latter is defined. If the second operand `<expr2>` is omitted, then the evaluation of the expression is safe, that is, GSL does not object (when this is feasible) to the result of the expression being undefined. This feature can be used in symbol definitions and substitutions to make GSL accept an undefined expression. See the description of these instructions for details.

The safe comparative operators return the same result as their equivalent comparative operators when both operands are defined. If one or both operator is undefined, the safe operators return FALSE while the normal operators produce an error. Notice that `a ?<> b` returns TRUE if both a and b are defined and they are not equal and FALSE otherwise.

The if operator returns the second operand if the first operand evaluates to a non-zero number. Otherwise the result is undefined. Thus an expression such as

```
test ?? "YES"
```

returns YES if test is 1 (or any other non-zero number); otherwise the result is undefined. The if operator can thus be combined with the default operator:

```
test ?? "YES" ? "NO"
```

If an operand is not a constant, then its type depends its value; if it looks like a number, then it is treated as a number, otherwise it is treated as a string.

Generally, additive, multiplicative and logical operators only apply to numeric operands. There are two cases where an arithmetic operator can apply to string values:

`+ "ABC" + "DEF"` evaluates to `"ABCDEF"`

`* "AB" * 3` evaluates to `"ABABAB"`

Substituting Symbols and Expressions At almost any place in a GSL script, you may use a substitute construct in the place of literal text. The format of a substitute construct is:

```
$( <expression> [% format] [: pretty-print] )
```

The construct is replaced by the value of the expression, output according to the format and pretty-print modifiers, if they exist. The order of the format and pretty-print modifiers is not important.

If the expression ends with a default operator that has no second operand, and the value of the expressions result is undefined then the substitution resolves to an empty string.

If a format string is provided, it is used to format the result before continuing. The format string is similar to that used by the printf function in C. It must contain exactly one conversion specification, consisting of zero or more of the flags `#`, `0`, `-`, and `+`, an optional minimum field width, an optional precision consisting of a period (`.`) followed by an optional number, and a mandatory conversion specifier among the following: `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, `c` and `s`. The data are always converted to the appropriate type (one of long int, double, char or char *) for the

conversion string. Note that not all legal C format strings are allowed in GSL. See details of the C printf function for more details. :)

The pretty-print modifier specifies how case modification and replacement of certain characters takes place. The valid pretty-print modifiers (not case-sensitive) are:

`UPPER` UPPER CASE

`lower` lower case

`Neat` Neat Case Modification

`Camel` camelCase

`Pascal` PascalCase

`no` No case modification

`c substitute_non_alpha_to_make_c_identifier`

`cobol SUBSTITUTE-NON-ALPHA-TO-MAKE-COBOL-IDENTIFIER`

`justify` Text is left justified within available space

`left` Entire block is shifted left as far as possible by removing the same number of spaces from each line.

`block` Text over multiple lines is formatted into a block

More than one pretty-print modifier may be specified; they should be separated by commas.

If GSL is in ignore-case mode (see below), and a substition expression consists of a single identifier and no case-modifier is specified (c or cobol may still be specified), the case in which the identifier name is specified is used as an example to determine whether the case of the result should be modified to UPPER, lower or Neat. To over-ride this, either disable ignore-case mode or provide an empty pretty-print string.

Some examples: Assume the identifier `IDENT` has the value `A few words from our sponsors` and identifer `XXX` is undefined.

`$(XXX)` produces a run-time GSL error: `Undefined expression:  XXX`

`$(XXX?"Undefined") Undefined`

`$(XXX?) (empty string)`

`$(IDENT%30s) A FEW WORDS FROM OUR SPONSORS`

`$(ident:upper) A FEW WORDS FROM OUR SPONSORS`

`$(Ident) A Few Words From Our Sponsors`

`$(ident:c) a\_few\_words\_from\_our\_sponsors`

`$(IDENT:) A few words from our sponsors`

`$(1 + 1) 2`

`$(ident:justify) a few words from our sponsors`

And:

```
/*  $("Description:":block)\
                $(ident:justify,block%-8s)  */
```

Gives:

```
/*  Description:  a few                    */
/*                words                    */
/*                from our                 */
/*                sponsors                 */
```

What You Can Substitute

A substitution can appear at any place in a literal string (template line or string constant) or as an operand in an expression. It can also replace part or all of a single identifier in a data specification, but not a period (`.`) or member construct (`->`).

Some examples: Assume the identifier `IDENT` has the value `NUM` and identifer `NUM` has the value `1`.

- `$($(ident))` gives `1`
- `$((ident)).NAME` gives `1.NAME` (may used in another expression as an

identifer.)

- `$(ident)+1` gives `NUM1`
- `$((ident))+1` gives `2`

4.3 Internals

4.3.1 Internal Variables

GSL has a series of internal variables, described below, that influence its behaviour in various ways. These variables are held in the scope `gsl` and can be modified by an instruction of the form:

```
[gsl].xxxxx = yyyy
```

You can also set the initial value of these variables when starting GSL by using a command of the form:

```
gsl -xxxxx:yyyy somescript.gsl
```

4.3.2 Template and Script Modes

Lines of GSL may be either script lines or template lines. GSL has two different modes for distinguishing script from template lines. In template mode, lines are assumed to be template lines unless they begin with a period (`.`). In script mode, lines are assumed to be script lines unless they begin with a greater-than symbol (`>`).

GSL starts in one of these modes, depending on the manner in which is was invoked. If it was invoked using an XML file as an argument, it begins in template mode as it is assumed that the XML file is to be used as data for creating an output file. If GSL was invoked using a GSL file as an argument, it begins in script mode.

You can change between template and script mode with the `template` and `endtemplate` commands. See the description of these commands below for details.

4.3.3 Template Lines

The simplest template line is just text, which is copied verbatim to the current output file. If no output file has been opened, or if the last output file has been closed, the output is copied to the standard output.

The backslash (\) serves several special functions in a template line. Firstly, if the last character of an template line is a backslash, then the line is output with no line terminator; otherwise a line terminator follows the template line. Secondly, a backslash introduces one of three special character sequences: `\n`, `\r` and `\t` which are replaced by a line feed, carriage return and a tabulation character (TAB) respectively. Thirdly, a backslash followed by and other character is replaced by that character; this allows characters which would normally be interpreted as script commands to be output literally.

4.3.4 Script Lines

The script commands are described below.

If a script command line ends with a backslash (\), then the following script line is treated as a continuation of the current line.

4.3.5 Comments

There are three ways to include comments in GSL scripts. The first is to place a hyphen (-) as the first character of a script line, or following the point (.) in a template line. The second way is to place a hash (#) after a GSL command. Any characters following the hyphen are ignored by GSL. The third way is to enclose comment text (which may continue over more than one line) inside comment markers (/* and */) just as in C. However if GSL finds these characters in a template line (but not inside a substitution) it assumes that they are destined for output, so does not treat them as a comment.

Examples:

```
.- This entire line is a comment

.output "file"  # This is a trailing comment

.output /* This is an embedded
multi-line comment */ "file"

If this is a template line, then /* this is not a comment */

$("but "/* this is */)
```

4.3.6 Ignorecase

GSL has two modes which influence case-sensitivity of identifier names. In the first mode (ignore-case), GSL is case-insensitive regarding identifier names and instead as a guide to modifying the output string, as described above (`Case Sensitivity`). In the second mode (case-sensitive), GSL treats the case of identifier names as significant. You can change the behaviour by setting the value of the attribute `ignorecase` in the scope `gsl` to 1 for ignore case and 0 for case sensitivity.

4.3.7 Shuffle

GSL can help to keep code neat by enlarging or shrinking white space so that column numbers match as far as possible between the script and the output file. For instance, if the value of the identifier `X` is `ABCDEF`, then:

```
$(X)   .
```

evaluates to

```
ABCDEF .
```

but

```
$(X?"Undefined") .
```

evaluates to

```
ABCDEF .
```

The shuffle algorithm uses the value of the attribute `shuffle` of the gsl scope). It expands a block of white space no shorter than `shuffle` as much as necessary so that the text following the white space is output in the same column. It also shrinks white space down to a minimum of `shuffle` to make space for text preceeding the white space. If `shuffle` is zero, then shuffle is disabled. The default value of `shuffle` is 2; this is the value which produces the results shown above.

If the current output ends with a backslash, then the shuffle continues on the following line. Thus

```
$(X?"Undefined")\\
          .
```

evaluates to

```
ABCDEF    .
```

Shuffle can cause problems in some cases, for example when outputting literal text where the size of white space is important. In this case shuffle should be disabled by setting the value to zero.

4.3.8 COBOL

GSL helps you make neat COBOL code by automatically filling the first six characters of each line with the four-digit line number followed by two zeroes. To enable this function set the value of the attribute `cobol` in the gsl scope to 1.

4.3.9 Line Terminators

GSL uses as its line terminator the value of of the attribute `terminator` of the gsl scope. The default value is "`\n`" but it could also be set to "`\r\n`", for example.

```
\# for Windows batch scripts
\.terminator="\r\n"
```

4.3.10 Escape Symbol

GSL uses the backslash "\" as its default escape symbol, mainly due to its POSIX / C roots. This can be very annoying in templates that have a lot of backslashes. You can override the escape symbol by changing the `[gsl].escape` attribute, or using the `-escape:X` command-line switch.

Note that this takes effect for the next script loaded, so you cannot use this in a script to modify how that script itself is processed. You can use it before e.g. including a script.

4.3.11 Substitute Symbol

GSL uses the string `$(` to open a substitution sequence. You can use any string instead, by changing the `[gsl].substitute` attribute, or using the `-substitute:X` command-line switch.

Note that this takes effect for the next script loaded, so you cannot use this in a script to modify how that script itself is processed. You can use it before e.g. including a script.

Note also that the closing symbol is always `)` and you cannot override this.

4.3.12 Arguments

If GSL is invoked with a `-a` switch, all arguments following the first are treated as arguments to the first script, rather than as further arguments to GSL, as would otherwise be the case. So if you type:

```
gsl -a myscript.gsl localhost 80
```

GSL defines attributes `arg1` in the symbol table `switches` in scope `gsl` with value `localhost` and attribute `arg2` with value `80`. A GSL script can access these values with an instruction of the form:

```
echo switches.arg1
```

or in a loop:

```
n = 1
echo switches.arg$(n)
```

4.3.13 Predefined Identifiers

There are some identifiers whose value is maintained by GSL in the global space referred to by the predefined scopes `gsl` and `global`. They are defined as attributes of the global item.

`script` The name of the GSL script file currently being processed.

`filename` The name of the XML file being processed.

`outfile` The name of the current output file; undefined if there is none.

`line` The line number of the line currently being output to the output file.

`me` The name of the current application: GSL.

`version` The version of the current application.

`switches` A symbol table holding all the command-line switches present when GSL was invoked.

4.4 Built-In Functions

GSL provides many built-in functions and uses `modules` to group related functions. Functions are listed under their module name. Each function listing shows the arguments it accepts. Optional arguments are shown with square brackets '`[]`'.

If an optional argument is provided, then previous arguments must also be provided. For example, the function `directory.open([name], [error])` accepts two optional arguments. If the `error` argument is provided, then the `name` argument must also be provided. Some functions take no arguments. If a function is given an incorrect number of errors GSL will print an error on the console and terminate.

If the provided arguments are of the wrong type or otherwise incorrect, the functions will return an undefined result, which can be handled with the default operator and tested with the defined() function.

Some functions accept an optional parameter, listed as `error`. If the parameter is provided, and an error occurs, the associated error text will be placed in the parameter and can be used as shown in this example.

```
dir = directory.open(".", error_text) ?

if ! defined(dir)
   abort "E: cannot open directory: " + error_text
endif
```

4.4.1 Global Functions

```
alias (item)
    To be explained.

class (item)
    Returns the class name for the scope of item.

count  ([<scope> .] <child> [, <expr> [, [<alias>]]]):
    counts the number of children of the supplied or most recently opened scope
    of the given name.  If an expression is specified, it is treated as
    a condition to determine which children are counted.  In this case,
    a new stacked scope is implicitly defined while the condition is
    evaluated.  The name of this scope is the name of the XML item, unless
    an alias is specified.  For compatibility with earlier versions of GSL,
    if no alias is specified, then a second, unstacked scope called 'count'
    and referring to the same XML item is created.
    For example: count (ITEM, ITEM.NAME = "ABC")
```

```
    returns the number of children of the most recently opened scope whose
    attribute NAME has the value 'ABC'.

defined (value)
    True if value is defined.

first ()
    True if current item is first in list.

index (item)
    Return index in current selection.

item (item)
    Return item number in original list.

last ()
    True if current item is last in list.

macro (name)
    True if name is a defined macro / function.

name ()
    To be explained.

scope ()
    To be explained.

total ()
    To be explained.

which ()
    To be explained.
```

4.4.2 conv

MODULE: GSL/conv package

```
Class: Conversion Functions

    Function: conv . chr (arg)
                Converts ascii code to character.
    If the argument is a number between 1 and 255, inclusive,
    returns the character represented by the ascii code.
    If the argument is a number is outside the range,
    returns the empty string.

    Function: conv . number (arg)
                Converts numeric looking input to number value.
    If the argument looks like a number, returns the numeric value.

    Function: conv . ord (arg)
                Converts character to ascii code.
    If the argument is a string, returns the ascii
    code of the first character, only.
```

```
    Function: conv . string (arg)
                Converts input to string value.
    If the argument is defined, returns string value.
```

4.4.3 diag

MODULE: GSL/diag package

```
Class: Diagnostic Functions

    Function: diag . used ()
        Shows amount (bytes) of memory allocated.

    Function: diag . allocs ()
        Shows number of memory allocations.

    Function: diag . frees ()
        Shows number of memory deallocations. Should equal allocs.

    Function: diag . display (filename)
        Writes the contents of the memory allocation list to the specified file.

    Function: diag . checkall ()
                Checks all allocated memory blocks for corruption and terminates
   the program if any are found.

    Function: diag . raise (signal)
            Send a signal corresponding to the numeric argument to the gsl process.
See signal(2) for numeric signal values.

    Function: diag . animate (value)
                Control GSL finite state machine progress where applicable.
    Use non-zero numeric argument to enable and a zero to disable.

    Function: diag . console_set_mode (mode)
            Sets console display mode; the argument can be one of:

0 - Output text exactly as specified. (Default).
1 - Prefix text by "yy/mm/dd hh:mm:ss "
2 - Prefix text by "hh:mm:ss "
3 - Same as #2 but output is fully flushed
```

4.4.4 environment

MODULE: GSL/environment package

```
Class: Environment Functions

    Function: env . get (name)
                Returns value of environment variable ‘name‘, if it exists.

    Function: env . set (name,[value])
                If value is provided, sets environment variable ‘name‘ to ‘value‘.
    Otherwise, clears environment variable ‘name‘.
```

4.4.5 fileio

GSL provides three modules for dealing with directories and files; one directory module and two file modules, one for working with independent files and the second for working with files during a directory traversal. We will discuss the second set after the first because it will make more sense that way.

Abstractedly, the modules have functions for working on, working with, and finding out about directories and files.

In the first category, directories have the `create` and `delete` functions which make them appear and disappear, modulo file permissions and other errors. Files also have the same functions, but `create` is spelled `open`. In addition, files have functions to `rename` and `copy` them. An important note: while it is generally important to check for errors in most operations, these operations almost demand checking for errors. Use of the default operator and error parameter will be well rewarded with working programs.

The second set of functions deal with the "contents" of directories and files.

A directory's purpose is to contain other files (directories are also files of a particular type). The only content operation is `open`, which returns a 'directory entry' object that can be used to iterate through the directory contents.

Files are a little richer and have operations to open them and to read from or write to them and to control where in the file to read or write.

File IO always begin with an open call, which returns a file handle.

The file handle is used in all subsequent content operations on that file. When the text refers to operations that affect the `handle`, keep in mind that this is short hand for the longer 'operations that affect the file that the handle represents'. The file is actually what is being worked on.

When opening a file, the `mode` parameter, a single letter, states how you intend to use the file, whether for 'r'eading, 'w'riting, or 'a'ppending.

Reading can be done with file.read(handle, [error]). Writing is done in a corresponding manner. Reading can also be done with function `file.slurp`, which returns the contents of the file. It is a shortcut to a common operation.

A file handle maintains an internal current file offset, which is a byte offset from the beginning of the file tells it where the next read or write should occur. A file opened for reading or writing will start of with an offset of 0, whereas a file opened for append mode will start with an offset corresponding to the end of the file.

The offset changes to reflect any read or write operations on the file. This is actually more than one needs to know just to read or write a file. However, it is sometimes useful and necessary to skip around inside a file, which is what `tell` and `seek` do. The function `tell` returns the current offset and `seek` changes the offset.

The final set of file functions manipulate files, file names and file metadata.

4.4.6 Directory Iteration

As mentioned, previously, directories can be opened with the `directory.open` function, which returns a 'directory entry' object. The 'directory object' represents a tree structure with child elements corresponding to the contents of the directory and can be iterated with a for/endfor loop. A child element is either a 'directory entry' or a 'file entry', depending on the file type. Both file and directory entries have a name() function, which returns 'file' or 'directory', as appropriate.

The loop

```
for dir. as elt

endfor
```

will allow access to all child elements, but, of course, the loop could be limited to files with

```
for dir.file
endfor
```

or directories with

```
for dir.directory

endfor
```

During iteration, the File Entry functions can be called on current item. These are similar to their corresponding File functions but do not take a `handle` parameter. Note that the `open` function reads all directory entries at start so any changes to the file system until the next open call. Also, iteration is only defined over files and directories; non file or directory entries are ignored. The open call will fail if the target is not a directory or it cannot find any valid files or directories in the target directory.

The directory entry has the attributes:

- `path`
- `name`

and the file entry has the following attributes:

- `path`
- `name`
- `size`
- `time`
- `date`

Which return the appropriate values from the file (or directory, which is, of course, a file).

The following example shows some of the attributes in use:

```
dir = directory.open(".", error_text) ?

if defined(dir)
  for dir.file as f
      echo "file:$(f.name) has size: $(f.size)"
  endfor
else
    abort "Error: " + error_text
endif
```

Note that:

If the directory entry `name` attribute is changed, the actual directory name is also changed. However, this operation does not return an error and cannot be recommended.

The file entry's default attribute is `name` so `f.` is the same as `f.name`. Directory entries don't have this default attribute so it's only useful when working with file entries.

File.open returns a File Entry object, so some of the file operations can be shortened a bit. For instance, file.read(handle) could also be written as `handle.read()`.

MODULE: GSL/fileio package

```
Class: Directory

    Function: directory . open ([path],[error])
                  Opens directory at ‘path‘ for iteration.
      ‘path‘ is opened relative to current directory.
      If ‘path‘ is not provided, uses the current directory.
```

```
      On success, returns a file entry for the first file in the directory.
      On error, returns an undefined value and sets ‘error‘, if provided.

      Note: In addition to permission, type, or existence errors, open will
      fail if the directory is empty since an empty directory has no entries.

    Function: directory . setcwd (path,[error])
                Changes current working directory to ‘path‘.
    Returns 0 on success.
    Returns -1 on error and sets ‘error‘, if provided.

    Function: directory . create (path)
                Creates directory ‘path‘.
    On success returns 0.
    On error, returns -1 (does not accept an error parameter).

    Notes:
    Can create multiple levels of directories, similar to 'mkdir -p' on unix.
    Will return success on directories already exist. Note that this is true
    even if the user does not access to said directory. Existence is all.
    If created, directories have permission 0775.

    Function: directory . delete (path,[error])
                Removes directory at ‘path‘.
    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.
    Note: will fail on non-empty directory.

    Function: directory . resolve (path,[separator])
              Locates ‘path‘ relative to current directory.

  If the path looks like an absolute directory, returns the cleaned up path.
  Otherwise appends the path to the current directory and returns the cleaned up result.

  Cleans-up the returned path by appending a '/' if necessary, and resolving any '..' subpaths.

  If ‘separator‘, a single character, is provided the resulting path components are separated by ‘sepa
  path components instead of the default separator (which depends on the operating system).


Class: File

    Function: file . open (filename,[mode],[error])
                  Opens ‘filename‘ with ‘mode‘ for reading or writing, depending on ‘mode‘.

      If mode is 'r', the file is opened for reading. Default if mode is not provided.
      If mode is 'w', the file is opened for writing. Empties file first.
      If mode is 'a', the file is opened for appending to existing content.

      On success, returns a file handle.
      On error, returns undefined and sets ‘error‘, if provided.

    Function: file . read (handle,[error])
```

```
            Reads a line from file ‘handle‘ .
On success, returns content read.
On error, returns undefined and sets ‘error‘, if provided.

Function: file . write (handle,string,[error])
            Writes ‘string‘ to file ‘handle‘.

On success, returns 0.
On error, returns -1 and sets ‘error‘, if provided.

Function: file . close (handle,[error])
            Closes file ‘handle‘.

On success, returns 0.
On error, returns -1 and sets ‘error‘, if provided.

Function: file . tell (handle,[error])
            Returns the current file offset of ‘handle‘.
The next ‘read‘ or ‘write‘ will start at that offset.

On success, returns offset, a number.
On error, returns undefined and sets ‘error‘, if provided.

Function: file . seek (handle,[offset],[error])
            Moves current file offset to ‘offset‘, a number, in file ‘handle‘.

If ‘offset‘ is -1, seeks to end of file, otherwise seeks
to specified offset in file.

On success, returns 0.
On error, returns -1 and sets ‘error‘, if provided.

Note: offset 0 is the beginning of the file.

Function: file . slurp (filename,[error])
            Reads the entire content of ‘filename‘.

On success, returns data read.
On error, returns undefined and sets ‘error‘, if provided.

Function: file . exists (filename,[error])
                Tests for file (or directory) existence.

    Returns 1 if the file exists.
    Returns 0 if the file does not exist and sets ‘error‘, if provided.

Function: file . timestamp (filename,[error])
                Returns the modification date and time of ‘filename‘,
    a file or directory name, as a number in the format YYYYMMDDHHMMSS.

    On success, returns a value.
    On error, returns -1 and sets ‘error‘, if provided.
```

```
    Function: file . rename (oldname,newname,[error])
                Renames file or directory ‘oldname‘ to ‘newname‘.
    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.

    Function: file . delete (filename,[error])
                Deletes file ‘filename‘.

    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.

    Function: file . locate (filename,[path],[error])
                  Searches for ‘filename‘ in a specified set of directories.
      If provided, ‘path‘ is first considered an environment variable
      and used to find the corresponding value. If there is no such value,
      ‘path‘ is used as is. If ‘path‘ is not provided, the environment variable
      ‘PATH‘ is used instead. If path is a literal, it is expected to be a list
      of directories separated by the default separator for the operating system.

      The current directory is implicitly prepended to the list and searched first.

      On success, returns the fully qualified path to the file.
      On error, returns undefined and sets ‘error‘, if provided.

    Function: file . copy (src,dest,[mode],[error])
                Copies file ‘src‘ to ‘dest‘ using ‘mode‘.

    ‘mode‘ determines how the file is copied;
    ’b’ for binary mode, ’t’ for text mode.
    The distinction only matters on DOS (and derivatives?).
    ’b’ is the default.

    On success, returns 1.
    On error, returns -1 and sets ‘error‘, if provided.

    Note: if ‘dst‘ exists, returns 1 and skips the copy operation.
    This could be considered a feature.

    Function: file . basename (filename)
                Removes dot and extension, if any, from ‘filename‘.
    If ‘filename‘ contains multiple extensions, removes the last one.

    Returns an appropriately shorn string or ‘filename‘, unchanged.


Class: Directory


Class: File

    Function: <file entry> . open ([mode],[error])
                Opens the file with ‘mode‘; ’r’, ’w’, or ’a’.
    On success, returns 0.
```

```
    On error, returns -1 and sets ‘error‘, if provided.

    Function: <file entry> . read ([error])
                Reads a line from file.
    On success, returns line read.
    On error, returns undefined and sets ‘error‘, if provided.

    Function: <file entry> . write (string,[error])
                Writes the string to file.
    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.

    Function: <file entry> . close ([error])
                Closes the file.
    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.

    Function: <file entry> . tell ([error])

    Function: <file entry> . seek ([offset],[error])
                Seeks to ‘offset‘. See File.seek for details.
    On success, returns 0.
    On error, returns -1 and sets ‘error‘, if provided.
```

4.4.7 gsl control

```
Class: GSL Control Class

    Function: gsl . include (filename,[template])
        Includes a GSL script file. Uses the current template mode unless
        over-ridden by the optional argument.

    Function: gsl . exec (command,[template])
        Executes a GSL script. If the script does not open an output file,
        its output is returned as the result of this function. Uses the
        current template mode unless over-ridden by the optional argument.
```

4.4.8 math

```
Class: Math Functions
    Function: math . abs (parm)
    Function: math . ceil (parm)
    Function: math . floor (parm)
    Function: math . mod (x,y)
    Function: math . rand ()
    Function: math . sqrt (parm)
    Function: math . exp (parm)
    Function: math . log (parm)
    Function: math . log10 (parm)
    Function: math . pow (x,y)
    Function: math . sin (parm)
    Function: math . cos (parm)
    Function: math . tan (parm)
    Function: math . sinh (parm)
```

```
    Function: math . cosh (parm)
    Function: math . tanh (parm)
    Function: math . asin (parm)
    Function: math . acos (parm)
    Function: math . atan (parm)
    Function: math . atan2 (x,y)
    Function: math . pi ()
    Function: math . asinh (parm)
    Function: math . acosh (parm)
    Function: math . atanh (parm)
```

4.4.9 regexp

```
Class: Regular Expression Functions
    Function: regexp . match (pattern,subject,[match])
```

4.4.10 process management

MODULE: GSL/process management package

```
Class: Process

    Function: proc . new (command,[workdir],[inname],[outname],[errname])
            Creates a process object.  The command is a native system command.  Does
not execute the command.

The ‘command‘ argument names a file to execute along with possible arguments.
The command will be searched for in the directories specified by the
PATH environment variable.

If ‘workdir‘ is provided, a chdir will be performed before the process is run.

If ‘inname‘, ‘outname‘, ‘errname‘ are provided,
the stdin, stdout, and stderr for the process will be
respectively redirected to the named files.

Returns the process object.


Class: Process handle

    Function: <proc handle> . setenv (name,[value])
            Sets an environment variable for the process.  Can only be called before
the process is started with proc_handle.run ()

    Function: <proc handle> . getenv (name)
            Gets an environment variable from the process.
Can only be called before the process is started with proc_handle.run ()

    Function: <proc handle> . run ([error])
            Runs a process created with proc.new ()

Returns -1 if there was an error creating the object.
Also places an error message into the parameter error.
```

4.4.11 script

```
Class: GSL Script Line
```

4.4.12 socket

```
Class: Socket
    Function: sock . passive (service,[error])
    Function: sock . connect ([host],service,[timeout],[error])

Class: Socket handle
    Function: <sock handle> . accept ([timeout],[error])
    Function: <sock handle> . close ([timeout],[error])
    Function: <sock handle> . read (buffer,[minimum],[timeout],[error])
    Function: <sock handle> . write (buffer,[timeout],[error])
```

4.4.13 string

```
Class: String Functions
    Function: string . length (string)
    Function: string . locate (haystack,needle)
    Function: string . locate_last (haystack,needle)
    Function: string . substr (string,[start],[end],[length])
    Function: string . trim (string)
    Function: string . justify (string,width,[prefix])
    Function: string . certify (number,[language])
    Function: string . replace (strbuf,strpattern)
        Parses the parameter 'strpattern' into a sequence of ',' separated
        replacements of the form 'search|replace', and then performs global
        replacement of those 'search' strings with the 'replace' counterpart.
        This function is limited to an output 4 times the size of the input strbuf.
    Function: string . search_replace (strbuf,strsearch,strreplace)
        Searches the parameter 'strbuf' for the first occurrence of 'strsearch',
        replacing it with 'strreplace'.
    Function: string . match (string1,string2)
    Function: string . prefixed (string,prefix)
    Function: string . prefix (string,delims)
    Function: string . defix (string,delims)
    Function: string . hash (string)
    Function: string . convch (string,from,to)
    Function: string . lexcmp (string1,string2)
    Function: string . lexncmp (string1,string2,count)
    Function: string . lexwcmp (string1,pattern)
    Function: string . matchpat (string1,pattern,[ic])
    Function: string . soundex (string)
    Function: string . cntch (string,value)
```

4.4.14 symb

```
Class: Symbol
```

4.4.15 thread

```
Class: Thread
    Function: thread . parse (command,[error])
```

```
        Parses the command, placing any error message in the parameter 'error'.
            Errors can also be recovered via thread class attributes.
            If successful, returns an object of type 'parsed item' which can
            then be run as many times as necessary, saving the need to re-parse
            each time.
    Function: thread . new (command,[error])
        Parses and runs the command, placing any error message in the parameter
            'error'.  Errors can also be recovered via thread class and child thread
            object attributes. If the parsing was successful, returns an object
            of type 'child thread'.
    Function: thread . sleep (csecs)
        Puts the current thread to sleep for the specified number of centiseconds.
            Returns -1 if csecs is invalid.
    Function: thread . receive ([receive])
        Receives a message from the thread's message queue.  If the queue is empty,
            the thread sleeps until a message is available.
            Places a reference to the sending thread in the first parameter.  Places
            arbitrary values from the message into successive parameters.

Class: Thread
    Function: <remote thread> . send ()
        Sends a message with an arbitrary number of arguments to the thread.

Class: Thread
    Function: <child thread> . interrupt ()
        Shuts down the thread.
    Function: <child thread> . send ()
        Sends a message with an arbitrary number of arguments to the thread.

Class: Thread
    Function: <parsed item> . run ([error])
```

4.4.16 time

```
Class: Time Functions
    Function: time . picture ([time],[picture])
    Function: time . number (time)
    Function: time . now ([date],[time])
        Assigns date and time to the passed parameters.  Also returns the time.
    Function: time . diff (date1,time1,date2,time2)
        Returns the difference (in csecs) between two date/times.

Class: Date Functions
    Function: date . picture ([date],[picture])
    Function: date . number (date)
```

4.4.17 XML

```
Class: XML
    Function: XML . new ([name])
        Creates a free `unattached` XML item.  In reality it is attached to the
            global variable ancestor, but only so that GSL can de-allocate it on
            termination.
```

```
            Therefore if the GSL script does not deallocate the item, it will remain
            until GSL shuts down.

    Function: XML . load_string (string,[error])
        Loads the supplied string as an XML item.
            Returns a pointer to the (first) resulting XML item or undefined if there
            was an error.  In the latter case, an error message is placed in the
            parameter ‘error‘ and in the XML thread context.

    Function: XML . load_file (filename,[error])
        Loads the file with the supplied name as an XML item.
            Returns a pointer to the (first) resulting XML item or undefined if there
            was an error.  In the latter case, an error message is placed in the
            parameter ‘error‘ and in the XML thread context.

Class: XML item
    Function: <XML item> . deleted ()
        Returns TRUE if the XML item has been deleted.
    Function: <XML item> . prev ()
        Returns the previous XML item.
    Function: <XML item> . string ()
        Returns the XML item formatted as a string.
    Function: <XML item> . load_string (string,[error])
        Loads the supplied string as a child of the item.
            Returns a pointer to the (first) resulting XML item or undefined if there
            was an error.  In the latter case, an error messa   ge is placed in the
            parameter ‘error‘ and in the XML thread context.
    Function: <XML item> . load_file (filename,[error])
        Loads the file with the supplied name as a child of the item.
            Returns a pointer to the (first) resulting XML item or undefined if there
            was an error.  In the latter case, an error message is placed in the
            parameter ‘error‘ and in the XML thread context.
    Function: <XML item> . save (filename,[error])
        Saves the XML item to a file with the given name.  Any file errors are
            places in the parameter ‘error‘ and in the XML thread context.  Returns
            zero if no error occurred, errno otherwise.

Class: XML value
```

4.5 Script Commands

4.5.1 Output File Manipulation

.output

```
.output <filename>
```

Closes the current output file, if one is open, and opens a new one.
Examples:

```
.output "myfile.c"
.output FILENAME
```

Where FILENAME is an identifier whose value is the desired file name.
.append

```
.append <filename>
```

Closes the current output file, if one is open, and opens a previously existing one and prepares to extend it. See the description of the output command for examples.

.close

```
.close
```

Closes the current output file, if one is open.

.literal

```
.literal [ from <filename> | " <text> " | << <terminator> ]
```

Copies text directly to the output file, without substition. The text can come from another file, a GSL expression, or from lines in the script, ending with a line beginning with the specified terminator.

Examples:

```
.literal from "file.txt"

.literal "whatever you want"

.literal << .endliteral
Lines are now copied without substitution of
things like $(abc).
.endliteral
```

4.5.2 Control Structures

.for

```
.for [[<data-specifier>] .] <name> [as <alias> | noalias] [nostack] [where <expr>] [by <expr>]
.for  [<data-specifier>] .         [as <alias> | noalias] [nostack] [where <expr>] [by <expr>]
```

Opens a scope and introduces a loop. The following block of code is processed once for each item specified. If no scope if specified, the most recently opened scope is assumed. The items processed are those children of the XML item corresponding to this scope. If the first form is used only children with the specified name are processed; if the second form is used, all children are processed.

The alias allows you to give the new scope a name other than the specified item name; use this when you nest scopes which would otherwise have the same name or to supply a scope name when using the second form.

The `where` clause allows you to specify a condition which must be satisfied for the code to be processed; the expression is evaluated before any processing occurs.

The `by` clause allows you to sort the items according to the result of evaluating the expression for each item. If no by clause is specified the items are processed from the oldest to the youngest, the same order in which they are described in the XML file.

The expressions in the `where` and `by` clauses are evaluated within the new scope. This means that they can access attributes of the iterating item.

During the evaluation of the `by` and `where` expressions, as well as during the processing of the code, the function `item (name)` returns the number of the child (1, 2, ...) of the current item. This number is associated with the XML item itself and is not affected by a `by` or `where` clause.

Within the loop, but not within `by` and `where` expressions, the function `index (name)` returns the index of the current iteration. This is associated with the loop, so that it always takes consecutive values.

If there are no items to iterate, an optional .else clause is executed.

.endfor

```
.endfor [<scope>]
```

Terminates a .for loop, closing the scope. The scope name is optional and does not affect the operation. GSL confirms that its value is the name of the scope to be closed and reports an error if this is not the case. In this way, GSL can be made to validate nested .for loops for you.

Examples:

```
.for RECORD.FIELD by NAME
$(FIELD.NAME)
.endfor
```

Outputs the names of the fields of the current record, sorted in alphabetical order.

```
.for FIELD as PASTURE where item () = 2
something
.else
somethine else
.endfor PASTURE
```

Processes only the second item named FIELD, and executes an .else clause if there is no such item.

.if

```
.if <expr>
```

Starts conditional processing of the following block of code if the result of evaluating the expression is non-zero.

.elsif

```
.elsif <expr>
```

May follow an `if` construct. Any number of `elsif` constructs may be used.

.else

```
.else
```

May follow an `if` or `elsif` or a `for` construct. The following block of code is processed if the logical value of all the expressions is FALSE.

.endif

```
.endif
```

Terminates a conditional processing construct.

Examples:

```
.if NAME = "JAMES"
something
.elsif NAME = "JAIME"
something else
.else
everything else
.endif
```

.while

```
.while <expr>
```

Introduces a loop. The following block of code is processed repeatedly as long as the expression evaluates to a logical value of TRUE, that is not equal to zero. Expression evaluation takes place before the code is processed, so that the code will never be processed if the expression evaluates to FALSE the first time.

.endwhile

```
.endwhile
```

Terminates a `while` loop.

Examples:

```
.define I = 0
.while I < 5
loop iteration number $(I)
.I += 1
.endwhile
```

.next

```
.next [<scope>]
```

Inside a `for` or `while` loop, causes immediate iteration, skipping execution of any code between the `next` command and the `endfor` or `endwhile` statement. If the scope is specified, then the `for` loop corresponding to that scope is iterated.

.last

```
.last [<scope>]
```

Inside a `for` or `while` loop, causes the loop to terminate iteration immediately. Control passes to the line following the `endfor` or `endwhile` statement. If the scope is specified, then the `for` loop corresponding to that scope is terminated.

4.5.3 Scope Manipulation

.scope

```
.scope <data-specifier> [as <alias> | noalias] [nostack]
```

Opens a new scope corresponding to the specified data.

.endscope

```
.endscope [<scope>]
```

Terminates a block opened with a `.scope` command, closing the scope. The scope name is optional and does not affect the operation. GSL confirms that its value is the name of the scope to be closed and reports an error if this is not the case. In this way, GSL can be made to validate nested `.scope` blocks for you.

4.5.4 Symbol Definition

```
.<data-specifier> [<operator>]= [ <expr> ]
```

Defines or undefines an XML attribute or item value. There are several different forms, described below:

If the scope is omitted from the data specification, GSL searches stacked scopes, from inner to outer, for one in which an attribute of the specified name exists. If none is found, it uses the outermost stacked scope, which effectively makes the identifier a global variable.

If the expression is left empty, then the symbol becomes undefined. If the expression ends with a default operator `?` but no default expression, then an undefined expression causes the symbol to become undefined rather than producing a runtime error.

If an arithmetic or default operator is specified, then the value assigned to the symbol is the result of that operator and the supplied expression to the former value of the operator.

Examples:

```
.x = 1
```

Assigns the value 1 to the identifier x in the most recently opened open scope where x is already defined, or in the global scope if x is undefined.

```
.->child. = "Value"
```

Assigns the string `Value` to the value of the first child of the innermost stacked scope.

```
.x *= 2
```

Multiplies the value of the identifier x by 2.

```
.x ?= y ? z ?
```

Does nothing if `x` is already defined; otherwise assigns it the value of `y`, or if `y` is undefined, then the value of `z`, or if `z` is undefined, `x` remains undefined.

4.5.5 Structured Data Manipulation

.new

```
.new [[<data-specifier>] . <name>] [before <before-scope> | after <after-scope>] [as <alias> | noalias]
.new <name> [to <data-specifier> | before <before-scope> | after <after-scope>] [as <alias> | noalias]
```

Creates a new XML item. This allows you to build new items in the data tree. The new item has the specified name and is a child of the XML item corresponding to th specified scope, or the most recently opened scope if none is specified. If a `before-scope` or `after-scope` is specified, then it must be the name of an open scope corresponding to a child of `<data-specifier>`, and the new item is inserted just before `<before-scope>` or just after `<after-scope>`; otherwise the new item is inserted after any existing children. The construct creates a new scope with the name specified by the alias or the item name if there is no alias. The following block of code is processed exactly once within this new scope. It would typically set some attributes of the new XML item. These values can then be retrieved during a future iteration of a `for` construct through the new item.

.endnew

```
.endnew [<scope>]
```

Terminates a `new` construct. The scope name is optional and does not affect the operation. GSL confirms that its value is the name of the scope to be closed and reports an error if this is not the case. In this way, GSL can be made to validate nested .new blocks for you.

Examples:

```
.new RECORD.FIELD
.    define FIELD.NAME = "NEW FIELD"
.endnew

.for RECORD.FIELD as OLDFIELD where NAME = "OLD FIELD"
.    new RECORD.FIELD before OLDFIELD
.        define FIELD.NAME = "NEW FIELD"
.    endnew
.endfor
```

.delete

```
.delete <data-specifier>
```

Deletes the data item corresponding to the specified scope. Once the item has been deleted, any attempt to reference it produces an error.

Examples:

```
.for RECORD.FIELD where TYPE = "COMMENT"
.    delete FIELD
.endfor
```

.copy

```
.copy [<from-scope>] [ to <parent-data> | after <after-scope> | before <before-scope> ] [as <name>]
```

Makes a copy the of XML item associated with `<from-scope>` (or the most recently opened scope if not specified) at the point specified by either the new parent (`to`) or new sibling (`after` or `before`), or as a child of the XML item of the most recently opened scope if no parent of sibling is specified. The `as` clause allows you to the new item to have a different name from the old item.

Examples:

```
.for DATABASE.TABLE
.    for RECORD.FIELD
.        copy FIELD to TABLE
.    endfor
.endfor
```

.move

```
.move [<from-data>] [ to <parent-data> | after <after-data> | before <before-data> ] [as <name>]
```

Re-attaches a data item at the point specified by a `to`, `after` or `before` clause, renaming it to the name specified in the `as` clause, if specified.

GSL detects any attempt to make an item its own descendent.

Note that moving an item does not invalidate any scope associated with it. If the moved item is associated with a future iteration of a `for` loop, the iteration will still take place even if the item is no longer a child of the extended scope from the `for` instruction.

Examples:

```
.for TABLE.RECORD
.    for RECORD.FIELD
.        move FIELD to RECORD
.    endfor
.endfor
```

.sort

```
.sort [[<data-specifier>] .] [<name>] [as <alias>] by <expr>
```

Sorts the specified items. A scope is created with each item in turn and is used to evaluate the expression. The result is then used to sort the items. The `as` clause allows you to give the created scope a different name. After execution, the specified items are in order and after any other children of the same parent.

4.5.6 Script Manipulation

.include

```
.include <filename>
```

Includes another script file. Deprecated - see `gsl`

.gsl

```
.gsl [ from <filename> | <expr> ]
```

Interprets the contents of the specified file or expression as GSL, just as though it were part of the script.
Examples:

```
.gsl from "header.gsl"
.gsl GSL.TEXT
```

.template

```
.template (0 | 1)
```

Turns template mode on or off.

.endtemplate

Terminates the block introduced by a `template` instruction.

4.5.7 Macros and Functions

Macros and functions are pieces of GSL which can be invoked with parameters. The only difference between a macro and a function is that macros are interpreted in template mode and functions in script mode.

Just like other data, macros and functions are attached to scopes, and can only be used within that scope.

When a macro or function executes, an unstacked scope is opened with the same name as the macro or function. An unstacked alias to this scope called `my` is also created. Parameters accessed via the `my` alias are flattened and parameters accessed via the macro or function name scope are not.

The macro or function-named scope and the `my alias` can also be used for local variables, thus enabling full recursion.

.macro

```
.macro [global .] <name> [(<param> [, <param>] ...)]
```

introduces a macro definition with the specified name.

.endmacro

```
.endmacro
```

Terminates a macro definition.

.function

```
.function [global .] <name> [([<param>] [, <param>] ...)]
```

Introduces a function definition with the specified name.

.endfunction

```
.endfunction <name>
```

Terminates a function definition.

.return

```
.return [<expression>]
```

Calling a function

```
.[<scope> .] <function-name> [([<expr>] [, [<expr>]])] ...)]
```

A macro or function can also be invoked as an expression. In this case, the expression value is that which is returned, or is undefined if there is no `return` statement.

This creates a special scope with the name of the macro or function, and attributes corresponding to the values of the parameters. This scope does not count in numeric scope specifications and cannot have children. It can be used to define local variables, but must in this case be specified by name.

The number of expressions (or empty expressions) must match exactly the number of parameters in the definition. An empty expression or an expression whose value is undefined causes the corresponding parameter to be undefined during processing of the macro code.

Examples:

The script

```
.macro echotwice (text)
.    echo echotwice.text
.    echo my.text
.endmacro
.
.echotwice ("Hello World!")
```

produces

```
Hello World!
Hello World!
```

The script

```
.function increment (value)
.    return my.value + 1
.endfunction
.
.increment (5)
```

produces

```
6
```

The script

```
.function incrementbyref (n)
.    $(my.n) = $(my.n) + 1
 .endfunction
.
.global.counter=5
.incrementbyref ("counter")
.echo global.counter
```

produces

```
6
```

The script

```
.function recursive (N)
.    echo my.N
.    my.localvar = my.N - 1
.    if (my.localvar > 0)
.        recursive (my.localvar)
.    endif
.    echo my.localvar
.endfunction
.
.recursive (3)
```

produces

```
3
2
1
0
1
2
```

4.5.8 Miscellaneous

.echo

```
.echo <expr>
```

Outputs the given expression to the standard output.

.abort

```
.abort <expr>
```

Outputs the given expression to the standard output and halts GSL operation.

4.5.9 Examples

See examples in Examples directory

This documentation was generated from gsl/README.txt using Gitdown