# `<bigwig>` Version 1.3 Tutorial

Claus Brabrand

BRICS Notes Series NS-00-5

ISSN 0909-3206 September 2000

# `<bigwig>` Version 1.3 Tutorial

Claus Brabrand
`brabrand@brics.dk`

September 2000

# <Tutorial>

The tutorial is based on a number of example programs illustrating the features of the <bigwig> language.

**Introduction** [page 2]

This section briefly introduces the individual aspects of the <bigwig> language.

**Getting Started** [page 5]

This section explains the session-based runtime model, which constitutes the foundation of <bigwig>.

**Dynamic Documents** [page 15]

Dynamic Documents allow easy and efficient creation of HTML documents. Document templates are treated as first-class values which can be combined using a "plug" operator and shown to the client with the "show-receive" statement.

**PowerForms** [page 34]

PowerForms is the <bigwig> solution to client-side form input validation.

**Database** [page 43]

This section describes <bigwig>'s relational database. <bigwig> contains a simple internal database; in a future version, external databases (e.g. MySQL) will be integrated.

**Concurrency Control** [page 56]

With many session threads running concurrently, there is a strong need for concurrency control on the server-side. The <bigwig> approach allows the programmer to specify concurrency constraints in a high-level logic notation.

**Macros** [page 65]

The syntax-based macro mechanism provides a method for constructing language abstractions in all the sublanguages of <bigwig>. For instance, complicated concurrency constraints can be wrapped into user-friendly syntax using macros. Macro Library Tutorials:

- The Standard Macro Library Tutorial: std.wigmac [page 82]
- The SQL Macro Library Tutorial: sql.wigmac [page 86]

# <Introduction>

[ Introduction | Runtime System | Dynamic Documents | Power Forms | Database | Concurrency Control | Security | Macro Mechanism | Availability ]

## Introduction

<bigwig> is a high-level programming language for developing interactive web services. Complete specifications are compiled into a conglomerate of lower-level technologies such as CGI-scripts, HTML, JavaScript, and the HTTP Authentication Protocol all running on top of a runtime system. <bigwig> is an intellectual descendant of the MAWL project but is a completely new design and implementation with vastly expanded ambitions.

The <bigwig> language is really a collection of tiny domain-specific languages focusing on different aspects of interactive web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, <bigwig> has the look and feel of C-programs but with special data- and control-structures.

**Paper:** "*<bigwig> - a Language for Developing Interactive Web Services*" (*submitted for publication...*).

## Runtime System

A <bigwig> service executes as a standalone process, communicating with the HTTP/CGI server is through a dedicated runtime system. This scheme overcomes the statelessness of the CGI-protocol in that the local state is present in the session thread for the duration of the service. Consequently, no local state image is ever required to be saved or restored. This is reminiscent of FastCGI, but with additional benefits, some of which are mentioned in the following. A service interaction can be bookmarked, paused, and resumed later. The browser's "back" button will function appropriately and not contain a sequence of obsolete html documents resulting from previous session interactions. Finally, the client is guaranteed response within 8 seconds. If the awaited html response page is not yet ready, a reason explaining the delay can be given. Also, concurrency control through a controller process (see Concurrency Control below).

See also the homepage for the Runtime System stand-alone package.

**Paper:** "*A Runtime System for Interactive Web Services*" (WWW8, Toronto, 1999)

## Dynamic Documents

HTML documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or attributes in tags. Such gaps may at runtime be plugged with concrete values. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. The documents are represented in a very compressed format, and the plug operations takes constant time only. A flow-sensitive type checker ensures that documents are used in a consistent manner.

**Paper:** *"A Type System for Dynamic Web Documents"* (POPL'00, Boston, 2000)

## Power Forms

The <bigwig> language has explicit support for form field validation. Regular expressions may be defined and associated with form input fields. These regular expressions are then compiled to efficient JavaScript (DFAs), exploiting only the subset of JavaScript that is known to run on all (normal) browsers. The result is that the input fields are validated incrementally on the client-side. An html page is only allowed to be submitted when all its input fields comply with their associated regular expressions.

See also the homepage for the PowerForms stand-alone package.

**Paper:** *"PowerForms: Declarative Client-side Form Field Validation"* (To appear in "World Wide Web Journal", 2000)

## Database

The familiar struct and array datastructures are replaced with tuples and relations which allow for a simple construction of small relational databases. These are efficiently implemented and should be sufficient for databases no bigger than a few MBs (of which there are quite a lot). A relation may be declared to be external, which will automatically handle the connection to some external server. An external relation is accessed with (a subset of) the syntax for internal relations, which is then translated into SQL.

## Concurrency Control

A <bigwig> service executes a dynamically varying number of threads. To provide a means of controlling the concurrent behavior, a thread may synchronize with a central controller that enforces the global behavior to conform to a regular language accepted by a finite-state automaton. That is, the 'control logic' in <bigwig> consists of finite-state automata. The controlling automaton is not given directly, but is computed (by the MONA system) from a collection of individual concurrency constraints phrased in first-order logic. Extensions with counters and negated alphabet symbols add expressiveness beyond regular languages.

**Paper:** *"Distributed Safety Controllers for Web Services"* (ETAPS/FASE'98, Lisbon, 1998)

## Macro Mechanism

An important mechanism for gluing these components together is a fully general hygienic macro mechanism that allows `<bigwig>` programmers to extend the language by adding arbitrary new productions to its grammar. All nonterminals are potential arguments and result types for such macros that, unlike C-front macros, are soundly implemented with full alpha-conversions. Also, error messages remain sensible, since they are threaded back through macro expansion. This allows the definition of Very Domain-Specific Languages that contain specialized constructions for building chat rooms, shopping centers, and much more. Macros are also used to wrap concurrency constraints and other primitives in layers of user-friendly syntax.

**Paper:** "*Growing Languages with Metamorphic Syntax Macros*" (*Submitted for publication...*, 2000)

## Availability

Version 1.3 is available as binary or source for UNIX/LINUX.

An overview of the `<bigwig>` language is presented in the paper "*`<bigwig>` - a Language for Developing Interactive Web Services*" (*submitted for publication...*).

# <Getting Started Tutorial>

[ Basics | Intermediate | Advanced ]

<bigwig> is a high-level programming language for developing interactive Web services. Complete specifications are compiled into a conglomerate of lower-level standard technologies such as CGI-scripts, HTML, JavaScript, and HTTP Authentication.

The aim of <bigwig> is to allow simple and inexpensive construction of advanced Web services. We are only concerned with specifying the behavior of the service; the graphical layout of the pages is left to other tools.

<bigwig> is committed to run on the lowest common denominator of the Web, that is, no special software is required for the browser or the server. Consequently, our services are CGI-based, use plain HTML for communication, and exploit only the subset of JavaScript that is common to the standard browsers.

The <bigwig> language is really a collection of individual domain-specific languages focusing on different aspects of interactive Web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, <bigwig> has the look and feel of C-programs with special data- and control-structures.

This tutorial will introduce the underlying runtime model, which in a sense constitutes the conceptual foundation of <bigwig>. This model builds upon a session concept, which allows sequential sessions with multiple interactions with the browser to be specified in ordinary programming notation.

## Basics

### Service structure

```
service {
  session S() {
    /* no statements */
  }
}
```

A <bigwig> program (a.k.a. *service*) starts with the keyword service, followed by the service specification enclosed in "{ ... }" braces. Sessions are the entry points to the service, much like the `main` routine in a `C` or `Java` program. However, unlike a `C` or `Java` program, a <bigwig> program may have more than one session. All sessions have associated URLs that when referenced (by a browser) cause an instance of the session to run, executing the code specified within the session's brackets. This service has one session named "S" with no statements in its body. When referenced, an instance of this session will immediately ``falls off the end" of the code, produces a default termination message onto the client's browser, and terminate.

---

## Shortest Hello World

```
service {
  session Hello() {
    exit <html>Hello World!</html>;
  }
}
```

This service has one session named "Hello" that when run executes a single exit statement, that sends an html document containing the text "Hello World!" onto the client's browser and terminates. A session thread is invoked by a client issuing a "GET" or "POST" CGI request using the URL mentioned in the getting started compiler page.

---

## Local declarations

```
service {
  session Add1() {
    int n; // int's are pr. default initially zero

    /* n has the value 0 */
    n++;
    /* n has the value 1 */
    exit (html) n;
  }
}
```

The session *Add1* has a (local) variable integer called *n*. By local (as opposed to shared) we mean that a session thread will (at runtime) have its own local (or private) variable accessible and visible to this session thread only. Variables in <bigwig> are automatically initialized (integers are initially zero). See the Initial Values section in the reference manual for more information. The first statement in this example increases the variable to the value one and the second (and last) statement exits this value onto the clients browser. Here, the information is exited by casting the value held in the variable *n* to an html expression. We shall see in the dynamic document tutorial how to present information to the client in a much nicer way. But to avoid getting into these details in this part of the tutorial, we shall for now present all information to the client in this way.

---

## Shared declarations

```
service {
  session Counter() {
    shared int n;

    /* n has some value */
    n++;
    /* n has a higher value */
    exit (html) n;
  }
}
```

This service is reminiscent of the previous one but with one important difference. The variable *n* declared in this example is prefixed with the type modifier shared. Shared variables are shared among all session threads. Thus, when one session thread updates it, subsequent reads in other sessions will get this latest written value. All `<bigwig>` services have their own (internal) database where shared variables are stored. (We are currently looking into integrating external databases).

---

**Two independent sessions**

```
service {
  session Counter() {
    shared int n;

    n++;
    exit (html) n;
  }

  session Hello() {
    exit <html>Hello World!</html>;
  }
}
```

This service has not one but two sessions, *Counter* and *Hello*. These two sessions are completely unrelated and could as well have been two entirely different services. Sessions are different starting points or ways of interacting with a service. A `<bigwig>` service can have any number of sessions. Typically a service has one or more sessions for the users and one for the administrators.

---

**Two collaborating sessions**

```
service {
  shared int n;

  session Counter() {
    n++;
    exit (html) n;
  }

  session Reset() {
    n = 0;
    exit <html>Reset!</html>;
  }
}
```

This service also has two sessions. These two sessions, however, are clearly related. They both manipulate the same shared integer variable *n*. The session *Counter* increases it by one and displays its value when run while the session *Reset* not surprisingly resets its value to zero when run. One can say that the two sessions *communicate* through the shared variable.

---

**Two-pass scope rules**

```
service {
  session Counter() {
    n++;
    exit (html) n;
  }

  shared int n;
}
```

The scoping rules in `<bigwig>` are two-pass meaning that variables are available at the same scope level even before the lexical point at which they were declared. However, the scope rules for variables declared in compound statements are one-pass (as in C and Java).

---

**Show (local state preserved)**

```
service {
  session Show42() {
    int n = 42;
    show (html) n;
    /* execution continues here after show
       with the entire local state as it was
       (`n' has the value 42) */
    exit (html) n;
  }
}
```

Instead of **exit**ing a document to the client, one can instead show a document to the client. The semantics is that the page will be shown to the client (in the browser) and execution will resume (with the local state preserved) after the show statement when the client submits the page. If the show document does not contain a "submit" button (as is the case with the document here), a default continue submit button will automatically be provided. When execution resumes after the show statement in the example, *n* will (still) have the value 42. Thus, the document exited in the final statement of the session will contain "42". This show construct plays a central role in the session concept, providing a means for interaction with the client.

---

**Show/Receive (interaction)**

```
service {
  session Interact() {
    int n;
    html Input = <html>
      Enter a number:
      <input type="text" name="number">
    </html>;

    show Input receive [n = number];
    /* Client's number is assigned to 'n'. */
    exit (html) n;
  }
}
```

The statement show-receive is fundamental in <bigwig> and provides the client-service interaction. The general pattern of a <bigwig> service (in fact, any interactive Web service) is to show a document to the client, receive some input, do something, show a reply page, receive some more data, do something, and so on). The service in this example will show a document prompting the client for a number which is received into a local variable *n* upon page submission. Hereafter, this entered value is exited onto the client's browser.

---

**Session calls**

```
service {
  shared int yes, no;

  html VoteDoc = <html>
    Do you think P=NP?
    Yes
    <input type="radio" name="answer" value="true">
    / No
    <input type="radio" name="answer" value="false">
  </html>;

  session Status() {
    exit (html) "yes: " + yes + ", no: " + no;
  }
  session Vote() {
    bool answer;

    show VoteDoc receive [answer = answer];
    if (answer) yes++;
    else no++;
    Status(); // Call session `Status'
  }
}
```

This vote service has two sessions *Status* and *Vote*, the second of which ends with a call to the first. The *Status* session can thus either be `run' initiated by a request to it on its own or through the *Vote* session.

---

**Session arguments**

```
service {
  int fac(int n) {
    if (n==0) return 1;
    else return n * fac(n-1);
  }

  session Factorial(int n) { // Session argument.
    if (n<0) exit <html>Negative number!</html>;
    exit (html) fac(n);
  }
}
```

As previously mentioned, a session is invoked through a CGI "GET" or "POST" request. A session can parameterized to take (CGI) *arguments* as in the example above. The argument names are explicit. If `http://*url_to_session*' designates the url to start this session, then `http://*url_to_session*&n=7' starts this session with argument *n* equal to 7. If argument *n* is missing it is assumed to have the initial value (which is zero for integers).

---

**Interacting with other services (get/post)**

```
service {
  session TalkToGoogle() {
    string s;

    s = post("http://www.google.com/")[q = "bigwig"];
    exit (html) s;
  }
}
```

For interaction with other services (<bigwig> or non-<bigwig> services), <bigwig> provides the functions get and post. They make a CGI request using respectively the "GET" and "POST" request method, to the URL specified in parentheses and with the list of arguments enclosed in square brackets. This example will make a "POST" request to "www.google.com" with argument "q" (query string) equal to "bigwig" and assign the result to the string variable *s*. Finally, the service will exit this string (cast to an html document) onto the client's browser. As one can see, it is easy to construct ``services talking to services".

## Security

```
ssl service { // All communication is encrypted.
  session S() {
    ...;
  }
}
```

A default <bigwig> service executes at a standard security level with protection against the most naive attacks. Tighter security levels of operation can be specified in <bigwig> by prepending the `service` or `session` keywords with the security modifier ssl. This modifier will cause all communication between the client and the server to be subject to the SSL cryptographic protocol. However, it is required that the web server supports SSL. Check out the security reference manual page for other security modifiers.

## Directory structure and garbage collection

```
service {
  session S() {
    file f;
    string filename = "foo.html";
    html H = ...;

    s = (string) H;
    f = open(dir + filename, "w");
    print(f, s);
    close(f);
    /* URL to the file: `url + filename'. */
    ...;
  }
}
```

When the session is started, it will have the BASEDIR appended with the service's name as its "current-directory". All files are opened relative to this directory. Each session thread is when started assigned a unique private directory that is automatically garbage collected when the session thread expires. The lifespan of a session (SPANDEFAULT) is set (in seconds) in the <bigwig> configuration file ".bigwig" (default is 48 hrs). This value can also be overridden at service, session, or individual show-level throgh the span modifier. This directory can be used to creates files of temporary nature that are automatically deleted when the session thread is expired. The expression, dir, designates a string file-system path to this directory (the expression, url, also designates this directory but as a URL).

# Intermediate

### Show/Receive/Timeout: (timeout clean-up)

```
service {
  shared bool locked; // initial value "false".

  session ExclusiveAccess() {
    string s;
    html H = ...;

    if (locked) exit (html) "Sorry, session in use!";
    locked = true;
    ...;
    show H receive [s = blah] timeout locked = false;
    ...;
    locked = false;
  }
}
```

A common problem with Web services in general is that a client may for some reason decide to abandon it in the middle of a show statement (that is, not submit the page). In <bigwig>, the session thread process will thus sleep on the server until it expires. When this happens, a **timeout**-statement can be executed as in the example. The example shows how to ensure that only one session thread is executing a certain region. When the timeout statement finishes, the session thread terminates.

Note however that the above does not guarantee that only one thread is in the region at a time as two sessions may evaluate the condition in the **if**-statement at precisely the same time. We shall in the concurrency control tutorial see how <bigwig>'s concurrency control language can be used to ensure that such a requirement will never be violated.

### Calling external (C-) functions from <bigwig>

```
                         ---foo.wig---

service {
  session EscapeToC() {
    string s;
    string my(int n); // function prototype

    ...
    s = my(87);
    ...
  }
}
                          ---bar.c---
```

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

char *my(int n) {
  return strdup("test");
}
                          ---shell---

%> gcc -c bar.c
%> bigwig bar.o foo
%> foo.install
```

If you specify a function prototype (a function declaration with no body), the <bigwig> compiler will assume that the implementation of the function will be linked with the <bigwig> C output when compiled to binary code.

**Note:** Make sure only to return (non-NULL) structures that are allocated on the heap as <bigwig> will free (garbage collect) the structure when it is no longer needed. This is currently possible for the following types:

| Language | Type | | | | | | |
|---|---|---|---|---|---|---|---|
| <bigwig> | bool | int | float | char | string | file | time |
| corresponding C type | int | int | double | char | char* | FILE* | time_t |

Another possibility of interacting with non-<bigwig> code is to use the built in **system** function.

# Advanced

### Flash (impatience handling)

```
service {
  session LongComputation() {
    html PleaseWait = <html>
      <h1>Factorizing large numbers</h1>
      This may take a while, please wait...
    </html>;

    flash PleaseWait;
    ...; // Factorize large numbers...
    exit (html) s;
  }
}
```

The statement flash is available for communicating information to the client when the session thread takes a long time to answer. As previously explained, each session thread is when it is started assigned a unique and private directory. This directory will contain a special file named "index.html" onto which all pages shown to the client by the session will go. If a show-statement is not executed within eight seconds after the client has submitted a request (on start-up or continuing a session), the client is automatically redirected to this page by a "connector" process. This page is automatically overwritten with a default excuse: "reply not ready yet, please wait...". The flash statement simply grants the service the possibility of asynchronously (with the connector) overriding this "index.html" reply file with a more informative excuse. Such an excuse page is always wrapped with some JavaScript causing it to reload every five seconds (the frequency is redefinable: refresh) until the ``real" answer is produced by the service, overriding this flash'ed page. If the service produces an answer within the eight seconds, the client never sees the flashed page. The technique itself is often referred to as ``client pull".

# <Dynamic Documents Tutorial>

[ Basics | Intermediate | Advanced ]

## Basics

Below are several different ways of writing the ubiquitous **Hello-World** service. These examples should provide basic insight on how html documents are constructed, combined, and shown to the client.

### Shortest Hello World

```
service {
  session Hello() {
    exit <html>Hello World!</html>;
  }
}
```

**Result:**

```
Hello World!
```

This is the shortest possible hello-world service. It has one session named *Hello* that when run simply exits a html document constant only containing the text "Hello World!" onto the client's browser after which the session terminates.

### Introducing an `html' variable

```
service {
  session Hello() {
    html D;

    D = <html>Hello World!</html>;
    exit D;
  }
}
```

**Result:**

```
Hello World!
```

Here, a variable *D* of type html is introduced. The first statement assigns to this variable an html document constant. The next (and last) statement of the session exits this document.

### Initialization of `html' variables

```
service {
  session Hello() {
    html D = <html>Hello World!</html>;

    exit D;
  }
}
```

**Result:**

```
Hello World!
```

The same as in the previous example, except the variable *D* is initialized to contain the hello-world document.

---

### Gaps and String plugging

```
service {
  session Hello() {
    html D, H;

    D = <html>Hello <[gap]>!</html>;
    /* string `World' is plugged into D
       and assigned to H. */
    H = D <[gap = "World"];
    exit H;
  }
}
```

**Result:**

```
Hello World!
```

In the first statement, we assign to *D* an html document containing a gap called "gap". The second statement plugs the constant string "World" into the document *D*, producing an entirely new document containing not surprisingly the text "Hello World!" which is subsequently assigned to *H*. Note that the expression *D <[gap = "World"]* does not side-effect D, but simply evaluates to a document value. The final statement exits *H* to the client.

---

### Gaps and Document plugging

```
service {
  session Hello() {
    html D = <html>Hello <[gap]>!</html>;
    html H;
    html W = <html><b>World</b></html>;

    /* html document W is plugged into D
       and assigned to H. */
    H = D <[gap = W];
    exit H;
  }
}
```

**Result:**

Hello **World**!

Instead of plugging a constant string "World", another document containing the text "World" in bold face is plugged into the document *D*. In this way, documents can be gradually composed and in a highly dynamic fashion (whence the name "Dynamic Documents" or "DynDoc"). Also, the programmer is no longer forced to do a linear construction from the first <html> to the last </html> tag.

---

### Plug evaluates to a document *value*

```
service {
  session Hello() {
    html D = <html>Hello <[gap]>!</html>;
    html W = <html><b>World</b></html>;

    /* the expression D <[gap = W] evaluates
       to a document that is exited to the
       client */
    exit D <[gap = W];
  }
}
```

**Result:**

Hello **World**!

This example emphasizes the fact that *D <[gap = W]* is (just) an expression that evaluates to a document value (which is exited to the client).

---

### Plug: Implicit coercion

```
service {
  session Hello() {
    int n = 42;
    html D = <html>Hello <[gap]>!</html>;

    exit D <[gap = n];
  }
}
```

**Result:**

```
Hello 42!
```

The plug expression will in fact implicitly coerce any type to `html' documents. For instance, the integer value *42* is converted to a string "42" immediately before being plugged. The same thing goes for bool, floats, chars, and time.

---

**Plug: Character escaping**

```
service {
  session Hello() {
    string s = "<b>World</b>";
    html D = <html>Hello <[gap]>!</html>;

    exit D <[gap = s];
  }
}
```

**Result:**

```
Hello <b>World</b>!
```

All strings plugged into `html' documents are properly escaped (The markup characters "<" and ">" are escaped to "&lt;" and "&gt;"). The reason for this is two-fold: to guarantee the safety of the analysis and for security reasons. Imagine what happens if a malicious client's input (say, "<script>location.replace("http://.../")</script>") gets propagated onto a document shown.

---

**Plug: Bypassing character escaping**

```
service {
  session Hello() {
    string s = "<b>World</b>";
    html D = <html>Hello <[gap]>!</html>;

    exit D <[gap = rawhtml(s)]; // caution!
  }
}
```

**Result:**

Hello **World**!

The predefined function rawhtml converts strings to html documents verbatim, leaving markup characters unchanged. Be cautious when using this function as it may introduce security breaches. Also, the analysis can no longer give all its static guarantees. When using this function it is the programmers responsability to make sure no unintended character sequences are introduced. Clearly, gaps cannot be dynamically plugged into documents this way, since the analysis is performed at compile-time.

---

### Attribute gaps (vs. html gaps)

```
service {
  session Hello() {
    html D = <html>
      <font color=[col]>Hello World!</font>
    </html>;

    /* the `col' gap cannot be plugged
       with documents */
    D = D <[col = "blue"];
    exit D;
  }
}
```

**Result:**

Hello World!

All the gaps we have seen so far have been the so called ``html gaps" where the syntax is <[*id*]> (where `*id*' is the name of the gap). The *col* gap is an example of another kind of gap, namely an ``attribute gap" - it is written within an html attribute (here "font"). Attribute gaps cannot like ``html gaps" be plugged with documents, only strings (or ints, floats, ...).

---

### Show: Local state preserved

```
service {
  session Hello() {
    html D = <html>Hello <[gap]>!</html>;
    html H = <html><b>World</b></html>;

    D = D <[gap = H];
    show <html>Click continue</html>;
    /* execution continues here after show
       with entire local state as it was
       (`D' has a bold-faced `World' in it!) */
    exit D;
  }
}
```

**Result:**

Click continue ; Hello **World!**

Instead of exiting a document to the client, one can instead show a document to the client. The semantics is that the page will be shown and execution will resume (with the local state preserved) after the show statement when the client submits the page. If the page does not contain a submit button, a default continue submit button will automatically be provided. Note that this default continue button is not shown in the "result" above. Thus, the document *D* exited in the final statement of the session, is the one constructed in the first statement.

---

### Input fields and Show-receive

```
service {
  session EnterName() {
    string s;
    html Input = <html>
      Name?: <input type="text" name="name">
    </html>;
    html Output = <html>Hi <b><[name]></b>!</html>;

    show Input receive [s = name];
    exit Output <[name = s];
  }
}
```

**Result:**

Name?: foo ; Hi **foo**!

...assuming "foo" was entered.

Here the document *Input* contains an input field of type "text". When such a document is shown, its values must be received. Conversely, when fields are received, they are required to be in the document shown. The identifier immediately following the equal character (`='), names an input field in the document shown. The identifier immediately preceding the equal character (`='), names an lvalue in the service. This lvalue can be of any type (the text entered by the client will be appropriately coerced).

---

### Checkboxes

```
service {
  session Checkbox() {
    vector string v;
    html Input = <html>
      x? <input type="checkbox" name="c" value="x">
      /
      y? <input type="checkbox" name="c" value="y">
    </html>;

    show Input receive [v = c];
    ...;
  }
}
```

**Result:**

x? ☐ / y? ☐ ; ...

When a document contains more than one checkbox (as inferred by the analysis), such an input field group must be received into a vector (or a relation). A single checkbox can be received in a basic type. The same thing goes for select (multiple) fields. There are a number of requirements on the different field kinds, but most of them behave roughly as the "text" field in the previous example. See the form input table for more information.

---

**Shorthand: ``Plug, then assign''**

```
service {
  session Hello() {
    html D = <html>Hello <[gap]>!</html>;
    html H = <html><b>World</b></html>;

    D =<[gap = H]; // plug, then assign (to D)
    exit D;
  }
}
```

**Result:**

Hello **World**!

The assignment expression *D = D <[gap = H]*, can be abbreviated to *D =<[gap = H]*, using the side-effecting ``*plug, then assign*''-operator "=<[". Note the three characters "=", "<", and "[" form one lexical token (that is, they cannot be separated by whitespaces). The syntax is reminiscent of the `+=' (add, then assign) operator from C/Java/<bigwig>.

---

**Iteration and `html' documents**

```
service {
  session IterateList() {
    int i;
    html H = <html><[more]></html>;
    html Item = <html><li><[no]><[more]></html>;

    for (i=3; i>0; i--) {
      H =<[more = Item <[no = i]];
    }
    exit H;
  }
}
```

**Result:**

- 3
- 2
- 1

Documents can easily be iteratively constructed. This will often happen when presenting data from a vector to the client. Notice that initially the document *H* contains a gap "more". This is important since all documents (expressions) in the program must have the same set of gaps for all program flows [see the example on "implicit closing of gaps" below for the exact details]. The program would be rejected by the compiler if *H* did not initially (in the **for** loop) contain the gap "more".

---

### Code Gaps: Code Expressions

```
service {
  int x = 21;
  session CodeExp() {
    html H = <html>res = <[(x*2)]></html>;

    exit H;
  }
}
```

**Result:**

res = 42

Just before document *H* is exited, the code expression, `<[(x*2)]>` is evaluated, yielding "42" which is implicitly coerced to a string and inserted into the document which is shown to the client.

---

### Code Gaps: Code Statements

```
service {
  int x = 5;
  session CodeExp() {
    html H = <html>
      res = <[{int r = x+1; r*7;}]>
    </html>;

    exit H;
  }
}
```

**Result:**

res = 42

The same thing happens here, the only difference being that the code gap is not a code expression but a code statement. The last statement in a code statement is required to be a statement-expression (here "*r*7;*") whose result is the result of the code statements.

---

**Code Gaps: Scope Restrictions**

```
service {
  html D;

  session CodeExp() {
    int x = 21;
    html H = <html>res = <[(x*2)]></html>;

    D = H; // `x' passed ``out of scope'' to D.
    exit H;
  }

  session SomeOtherSession() {
    ...;
    exit D; // `x' ``out of scope''!
  }
}
```

**Result:** N/A, Illegal service.

The scope for variables in code expressions and code statements is the toplevel scope outside the sessions. Otherwise it would be possible to pass variables *out of scope* as happens with the *x* usage in the code gap in the example.

---

**Separating designer/programmer tasks by lexical inclusion**

```
service {
  session Hello() {
    html H = #include "../docs/hello.html";

    exit H;
  }
}
```

**Result:**

HELLO WORLD

Notice how the designer and programmer's tasks have been separated in the code. This division can, in fact, be made even more explicit by lexically including the documents required by the service:

Now, the designer is free to design the html page in for instance FrontPage or some other html page design tool. The document is automatically included in the service by the lexical analyser during service compilation.

---

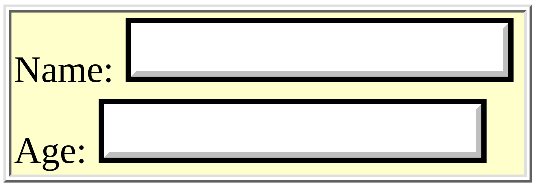
# Intermediate

### Rapid prototyping

```
service {
  session Hello() {
    html P =  <html>
      Name: <input type="text" name="N"><br>
      Age: <input type="text" name="A">
      <[more]>
    </html> @ ".../docs/person.html";

    exit P;
  }
}
```

**Result:** ...if the file "../docs/person.html" does not (yet) exist.

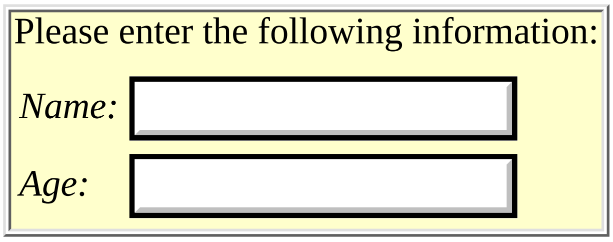

Name:
Age:

When the file designated in the *<html>...</html> @ "URL"* construction does not exist, the constant in-lined document is used.

```
                  ---../docs/person.html---

<html>
  Please enter the following information:
  <table>
    <tr>
      <td><em>Name:</em></td>
      <td><input type="text" name="N"></td>
    </tr>
    <tr>
      <td><em>Age:</em></td>
      <td><input type="text" name="A"></td>
    </tr>
  </table>
  <[more]>
</html>
```

**Result:** ...if the file "../docs/person.html" does exist.


Please enter the following information:

*Name:*

*Age:*

Whereas, when the document does exist (the file is non-empty) it is the one used. The two html documents are required to have the same flow types (i.e. same gaps and fields). This is verified by the compiler at compile-time.

The idea behind this construct is to provide a means for rapid prototyping and to aid collaboration between the *programmer* and the *designer* of a Web service. The *programmer* rapidly makes some prototype html documents, with focus on the functionality (the fields and gaps) and not on the graphical layout of the document. Then, as the *designer* finishes the ``real" documents, they replace the prototype ones.

---

**Auto-wrapped tags**

```
service {
  session Hello() {
    html H = <html>
      <head><title>foo</title></head>
      <body bgcolor=[col]>
        Hello World!
      </body>
    </html>

    exit H <[col = "yellow"];
  }
}
```

**Result:**

Hello World!

The tags "<**html**>" and "</**html**>" are in fact just delimiters saying that whatever comes in between is an html document template. They are not themselves part of the html document template. However, when an html document template...

```
<html>...</html>
```

...is **shown**, it is wrapped with an `html`, a `head`, a (default) `title`, a `body`, and a `form` tag with an appropriate continue "action" url, yielding:

```
<html>
  <head>
    <title><bigwig> service: hello</title>
  </head>
  <body>
  <form action="continue url">
    ...
  </form>
  </body>
</html>
```

Note that the <form> tag pair is not added when a page is **flash**ed or **exit**ed.

In order to be able to add information the placement of which is required before the <form> tag, `<bigwig>` intercepts an optional <body> tag and uses it, overwriting the default one (placing it immediately preceding the autogenerated <form> tag). As can be seen in the example, such a <body> tag can contain attributes (for instance, a bgcolor attribute as in the example). Everything written outside this <body> tag, gets placed outside the autogenerated <form> tag. Thus, the title of the document shown will be "foo". Being able to insert information outside the form (and body) is particularly useful when a page is to contain JavaScript code. Note that <body> tags along with the information preceeding them in documents plugged into other documents is discarded.

---

**Gaps are unordered**

```
service {
  session Address() {
    int n = 42;
    string s = "Somewhere street";
    html H;
    html UK = <html><[number]>, <[street]></html>
    html DK = <html><[street]> <[number]></html>

    if (...) H = UK;
    else H = DK;
    exit H <[number = n, street = s];
  }
}
```

**Result:**

42, Somewhere street , when `...' evaluates to *true*

Somewhere street 42 , when `...' evaluates to *false*

The plug operator can perform multiple pluggings in one go, as is evident in the plug expression *H <[number = n, street = s]* above. Each document in `<bigwig>` has a *set* of unordered gaps (i.e. *not* a sequence of gaps). The fact that the order of gaps is inconsequential, can be exploited to customize information presentation. For instance, in Denmark addresses are written street name followed by house number, whereas in the United Kingdom they are the house number followed by a comma and the name of the street. The plug operation will plug *n* and *s* into the gaps *number* and *street* regardless of their respective placements in the document *H* in which we plug.

---

**Implicit closing of gaps**

```
service {
  session ImplicitClose() {
    html H = <html>Hello <[what]>!</html>

    if (...) H = H <[what = "World"];
    /* Here, H no longer has the what gap */
    exit H;
  }
}
```

**Result:**

Hello World!

After the if-statement, document *H* no longer has the *what* gap. The reason is that its presence is not guaranteed (if, for instance, the expression `...' evaluates to true). Consequently, the *what* gap is implicitly closed in the ``else branch'' of the if. Implicit closing happens automatically so that the service will obey the flow join requirements dictated by the dynamic document analysis. Attempts to plug in *H*'s *what* gap after the **if**-statement will be denied by the compiler, yielding a compile-time error.

---

**A gap may not occur twice in the same document**

```
service {
  session NoMultGaps() {
    html B;
    html D = <html>Hello <[what]><[g]></html>;
    html H = <html>World <[what]></html>;

    B = ~~D <[g = H]~~; // Illegal document!
    exit B;
  }
}
```

**Result:** N/A, Illegal service.

Our analysis and implementation prohibits a gap from being present twice in a document. Consequently, the document produced by the plug operation (with two gaps by the name of *what*) above is illegal and the service will be rejected by the compiler, yielding a compile-time error.

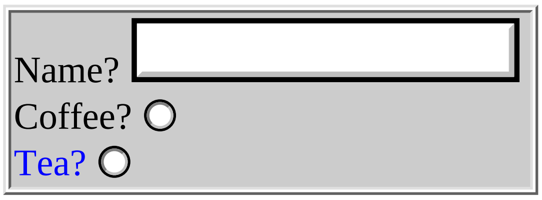
---

**Analysis inference (track)**

```
service {
  session OrderDrink() {
    string order;
    html D;
    html H = <html>
      Name? <input type="text" name="name">
      <br>
      <[choices]>
    </html>;
    html Coffee = <html>Coffee?
      <input type="radio" name="drink"
             value="coffee">
      <font color=[col]><[alts]></font>
    </html>;
    html Tea = <html><br>Tea?
      <input type="radio" name="drink"
             value="tea">
      <[alts]>
    </html>;

    H = H <[choices = Coffee <[alts = Tea]];
    /* Here, track(H) would report:
       gaps = { ("col", attrGap),
                ("alts", htmlGap) }
       fields = { ("name", textField),
                  ("drink", radioField) } */
    D = H <[col = "blue"];
    show D receive [order = drink];
  }
}
```

**Result:**


Name?
Coffee?
Tea?

Clearly, the programmer needs to be aware of how gaps and fields propagate in the service. To help the programmer, we have added a keyword "track" which is a predefined function that is the identity on expressions of type html except for one fact: It will report, at compile-time, the flow-type of its html argument (as inferred by the flow analysis). A document's flow-type is its set of gaps and their kinds (attribute or html) and its set of fields and their kinds (text, radio, checkbox, ...). The function is really not part of the <bigwig> language, but is considered a helpful compiler feature.

---

**Functions and `html' documents**

```
service {
  session Hello() {
    html H = <html>Hello <[gap]>!</html>;

    html f(bool in_bold, string s) {
      html B = <html><b><[content]></b></html>;

      if (in_bold) return B <[content = s];
      else return (html) s;
    }

    exit H <[gap = f(true, "World")];
  }
}
```

**Result:**

Hello **World**!

Here we have defined a function *f* that takes a boolean *in_bold* and a string *s* as arguments and produces an html document. The document produced is *s* as an html document, and with the text in bold face if the first argument is *true*. The function contains a type-conversion expression *(html) s* (that casts *s* to an html document).

---

**Recursive functions and `html' documents**

```
service {
  session RecList() {
    html list(int n) {
      html Item = <html>
        <li><[no]><[more]>
      </html>;

      if (n==0) return <html></html>;
      return Item <[no = n, more = list(n-1)];
    }

    exit list(3);
  }
}
```

**Result:**

- 3
- 2
- 1

Needless to say, html documents work with recursion. This example is equivalent to the iterative example previously mentioned.

---

**Matching `html' documents**

```
service {
  session Dilbert() {
    html OutDoc = <html>
      <h1>Today's Dilbert</h1>
      <img src=[src] alt="today's Dilbert comic">
    </html>;
    string data = get("http://www.dilbert.com/");
    string src_str;
    match(data,<html><[]>
      <img src=[src] alt="today's Dilbert comic">
    <[]></html>)[src_str = src];
    exit OutDoc <[src=
        "http://www.dilbert.com"+src_str];
  }
}
```

**Result:** *Today's Dilbert image without any adds.*

For now, the second html argument to match must be a constant html document. The constant document contains two unnamed gaps "<[]>". Whatever such gaps match in a match operation is discarded. Note that in order to work properly, the return character and the spaces after the first and before the second unnamed gap should be removed.

---

## Advanced

Unfortunately, due to the undecidable nature of the analysis, there are many services that are unfairly rejected by the compiler. Such examples can be found below:

---

**Attribute and html gaps (continued)**

```
service {
  session Hello() {
    html D;
    html A = <html>
      <font color=[gap]>Hello World!</font>
    html>;
    html H = <html>Hello <[gap]>!</html>;

    if (...) D = A;
    else D = H;
    /* gap "gap" in D is an attribute gap
       (i.e. D can no longer be plugged with
       an html document). */
    exit D <[gap = "blue"];
  }
}
```

**Result:**

Hello World! , when `...' evaluates to *true*

Hello blue! , when `...' evaluates to *false*

After the if-else statement, when the flows of the statements from the if-branch and the else-branch meet, the gap named "gap" in document D becomes an attribute gap regardless of which of the two branches is taken at runtime. Thus, D can no longer be plugged with documents (otherwise there could be problems if the expression ... evaluated to true). [In formal analysis terminology: least-upper-bound of an attribute gap and an html gap is an attribute gap].

---

## Tuple fields

```
service {
  schema PersonSchema {
    bool married;
    string name;
  }

  html f(int n) {
    int i;
    html T = <html><tuple name=person>
      Name?: <input type="text" name="name">
      <input type="checkbox" name="married"
             value="true">
    </tuple><[more]></html>;
    html H = T;

    for (i=0; i<n-1; i++) {
      H =<[more = T];
    }
    return H;
  }

  session Tuples() {
    vector PersonSchema p;
    html D;

    D = f(random(10));
    show D receive [p = person];
    ...;
  }
}
```

**Result:** *Not shown!*

**Note**: *This example assumes knowledge of schemas.*
The fact that one has to explicitly receive every single field, is incompatible with creating a dynamic number of input fields for the client to fill in. This is what the <tuple> tag was added for. The <tuple> tag is internal to <bigwig> and never actually shown to the client.

---

**Analysis shortcomings (undecidability)**

```
service {
  session Checkboxes() {
    int n;
    string choice;
    html D = <html><[gap1]><[gap2]></html>
    html H = <html>
      <input type="checkbox" name="c"
             value="...">
    </html>;

    n = ...;
    if (n%2==0) D =<[gap1 = H];
    /* Here, it is assumed that D has a
       checkbox group `c' with one
       element. */
    if (n%2==1) D =<[gap2 = H];
    /* Here, it is assumed that D has a
       checkbox group `c' with more than
       one element. */
    show D receive [~~choice~~ = c];
  }
}
```

**Result:** N/A, Illegal program!

After the first if-statement, the two possible flows (the then-branch and the non-existant or empty else-branch) cause the analysis to assume that there is a checkbox group named *c* with one checkbox.

This is a fair assumption because a non-checked and a non-existant checkbox submit the same (non-existant) information upon submission. After the second if-statement the same thing happens again, forcing the analysis to safely assume that there could possibly be several checkboxes in the group named *c*. Consequently, when the document *D* is shown, it is assumed to potentially return several (more-than-one) units of information and is thus required to be received in a vector (or relation). This is why the above program will fail. It can be deduced that independent of the value of *n*, *D* could never hold two checkboxes. However, this deduction is due to a very complex inter-relationship between the two expressions *n%2==0* and *n%2==1* that a compiler could never hope to generally uncover. Precisely this example could be made detectable, but due to the undecidable nature of the problem itself, there would still be infinitely many similar undecidable problems, regardless of how sophisticated we make our analysis.

---

## Analysis shortcomings (monovariance)

```
service {
  session Id() {
    html H = <html>Hello <[gap]>!</html>;

    html id(html H) {
      return H;
    }

    H = id(H);
    H = H <[gap = "World"];
    H = ~~id(H);~~ // Illegal call!
    exit H;
  }
}
```

**Result:** N/A, Illegal program!

Since the interprocedural data-flow analysis on dynamic documents is monovariant (as opposed to poly-variant), there are some restrictions regarding the use of functions. It is required that the set of gaps and their kinds (attribute/html) and the set of fields and their kinds of all html arguments are the same for all calls to a function. The same thing goes for any html values returned. The first call to the id function dictates that from now on the argument (and result) given to id must be a document that has exactly one gap named "gap" of kind `html'. Consequently, the second call to id with a document without any gaps is denied by the compiler, yielding a compile-time error. This example would benefit from making the analysis poly-variant which it may be in the future.

---

# <PowerForms Tutorial>

[ Basics | Intermediate | Advanced ]

**Note**: This tutorial assumes basic knowledge on:

- Dynamic Documents

## Basics

This is the PowerForms tutorial that will explain how to automatically ensure that all input fields are appropriately filled in by the client prior to page submission. Related benefits include file scanning and string matching and will also be presented. As is evident below, our solution is based on regular expressions:

**Formats: Client-side input validation**

```
service {
  session EnterDigit() {
    format Digit = range('0', '9');
      /* Format declaration */

    int n;
    html H = <html>
      Enter your favorite digit:
      <input type="text" name="d"
             format="Digit">
    </html>;

    show H receive [n = d];
    ...;
  }
}
```

Formats are declared using the keyword format followed by an identifier that will henceforth refer to the format. A format is really a regular expression (regexp) specified in an elaborate syntax. The format *Digit* in the example is declared to match the ten characters in the range from '0' to '9' (including both end points). The format is subsequently bound to the text input field named "d" in the document *H* through the *format="Digit"* attribute. When run, the *EnterDigit* session will ask the client for his favorite digit and incrementally verify that the text entered in the text input field matches the regular expression (i.e. that the client enters a digit). The document can only be submitted when the input is matched by the regexp.

The validation status will be shown in the browser's status bar and by status icons placed next to the input field. By default these icons show traffic lights with the colors red, yellow, and green, signalling respectively that the format is invalid (not in the language defined by the regexp), on the way of becoming valid (in the prefix of the regexp), and valid (in the regexp).

---

## Perl style

```
service {
  session EnterDigit() {
    format Digit = regexp("[0-9]");
      /* Perl style format declaration */

    int n;
    html H = <html>
      Enter your favorite digit:
      <input type="text" name="d"
             format="Digit">
    </html>;

    show H receive [n = d];
    ...;
  }
}
```

This service is equivalent to the one above, but with Perl style regexps instead.

---

## Defining formats

```
service {
  session EnterAge() {
    int n;
    format Digit = range('0', '9');
    format Age = concat(Digit, star(Digit));
    html H = <html>
      Enter your favorite age:
      <input type="text" name="a"
             format="Age">
    </html>;

    show H receive [n = a];
    ...;
  }
}
```

Formats are not surprisingly allowed to be defined in terms of each other. Here the *Digit* format from the previous example is used in the definition of another format *Age* that will match any non-zero number of digits. Concat takes any number of regexps and is the concatenation of these. Star (a.k.a. Kleene's star) takes one regexp and is any number of repetitions (even zero) of this regexp. This format is bound to the text input field named "a" in the document *H*. When this document is shown, the client must and can only enter a valid age (a positive integer).

---

### Perl style

```
service {
  session EnterAge() {
    int n;
    format Digit = regexp("[0-9]");
    format Age = regexp("{Digit}{Digit}*");
    html H = <html>
      Enter your favorite age:
      <input type="text" name="a"
             format="Age">
    </html>;

    show H receive [n = a];
    ...;
  }
}
```

This service is equivalent to the one above, but with Perl style regexps instead. The syntax {*id*} designates another format named *id*.

---

### No circularly defined formats

```
service {
  session Enter() {
    format RecF = concat(star("foo"), ~~RecF~~);
    /* Error: Circular format! */

    ...;
  }
}
```

**Note**: This is an illegal service!

As with all other toplevel declarations, formats have two pass scope rules (they are visible on the same scope level even before their lexical definition point). However, formats cannot be circularly or recursively defined.

---

### Email format

```
require <std.wigmac>

service {
  session EnterEmail() {
    string s;
    format Digit = range('0', '9');
    format Alpha = range('a', 'z');
    format Word = plus(union(Digit, Alpha));
    format Email = concat(Word, "@", Word,
        star(concat(".", Word)));
    html H = <html>
      Enter your email:
      <input type="text" name="e"
             format="Email">
    </html>;

    show H receive [s = e];
    ...;
  }
}
```

This service defines four formats. The first is the *Digit* format we have already seen. The second, *Alpha*, is defined to be any lower case alphanumeric character. The third is any (non-zero) number of repetitions of the either a digit or a lower alphanumeric case character. The plus construct is really a regexp macro being invoked. The macro takes one regexp argument and is the concatenation of the argument with star of the argument. We shall see in the macro tutorial how this macro is defined.

---

**Escaping validation: ignoreformats**

```
service {
  session EnterEmail() {
    string s;
    format Email = ...;
    html H = <html>
      Enter your email:
      <input type="text" name="e"
             format="Email">
      <input type="submit" value="Cancel"
        ignoreformats>
    </html>;

    show H receive [s = e];
    ...;
  }
}
```

Normally, one cannot submit a page while the input fields are not all correctly filled in. Sometimes, however, it is nice to be able to disable this functionality which is exactly what the attribute ignoreformats does. The attribute is applicable to all input fields that causes the document to be submitted (that is, submit, continue, and image fields). The example will show a document prompting the client for his email, but the client has the possibility of pressing the cancel button (even if the email field is not correctly filled in).

---

### Customizing errors and warnings

```
service {
  session CustomErrors() {
    int n;
    format Digit = range('0', '9');
    format Number = plus(Digit);
    html H = <html>
      Enter your email:
      <input type="text" name="n"
             format="Number"
          red="Not a number!">
          yellow="Enter a valid number"
    </html>;

    show H receive [n = n];
    ...;
  }
}
```

As previously explained, the incremental validation status is shown in the browser's staus bar while the client is entering data. The status bar will feature standard default messages corresponding to the three states red, yellow, and green (mentioned in the first example). These messages can easily be redefined by assigning the corresponding attributes "red" and "yellow". The "red" and "yellow" messages are also the ones shown when the client attempts to submit a document containing data that violate formats.

---

### Changing the status icons

```
service {
  session EnterDigit() {
    int n;
    format Digit = range('0', '9');
    html H = <html>
      Enter your favorite digit:
      <input type="text" name="d"
             format="Digit">
    </html>;

    show H receive [n = d];
    ...;
  }
}
```

You can change the status icons to your own images reflecting the particular style and look-and-feel you want your service to have. This is done outside the service by placing four images "../powerforms/red.gif", "../powerforms/yellow.gif", "../powerforms/green.gif", and "../powerforms/na.gif" in your RGYIMAGEDIR (set in your ".bigwig" configuration file). If the changes are only relevant to the service at hand, you may want to consider having a local ".bigwig" configuration file in the service's directory (in which you compile). If you do not want any status icons, (for now) you have to make 1-pixel transparent images [sorry].

---

## Formats: String matching

```
service {
  session MatchString() {
    int n;
    string s;
    format Digit = range('0', '9');
    format Number = plus(Digit);

    s = ...;
    if (match(s,Number)[]) {
      /* if `s' matched `Number' */
      n = (string) s;
      ...;
    } else {
      n = -1; // `s' was not a number
      ...;
    }
  }
}
```

As in Perl, strings in `<bigwig>` can (at runtime) be matched against regexp formats. In the example, the string *s* is matched against the format *Number* (the empty square brackets are explained below). The match construction returns a boolean stating whether or not the string is in the language defined by the regular expression.

---

## String recording

```
service {
  session RecordString() {
    string d;
    string e = "bigwig@brics.dk";
    format Word = ...;
    format Email = concat(Word, "@",
      [domain = // format recording
        concat(Word,star(concat(".", Word)))]
    );

    if (match(e,Email)[d = domain]) {
      /* Here, `d' is "brics.dk". */
      ...;
    }
  }
}
```

The format *Email* in this example contains a ``recording" regexp named "domain". A recording regexp will record the string its regexp argument matches when used in a match contruction. Since the string *e* is matched by the regexp format *Email*, match evaluates to **true** and *d* is assigned the value "brics.dk". If a string is not matchable (match returns **false**), all recordings are assigned initial values (0 for integers, "" for strings, etc.). This format recording mechanism is reminiscent of parentheses in Perl regexps.

---

**Formats: File-scanning**

```
service {
  session FileScan() {
    int n;
    file f;
    format Digit = range('0', '9');
    format Number = plus(Digit);

    f = open("hello.txt", "r");
    n = scan(f, Number);
    close(f);
    ...;
  }
}
```

A final application of regexp formats is file scanning. The construction scan takes a file handle followed by a regexp format and scan as much of the file from the current file position that is in the regular language defined by the format and advance the file pointer accordingly. Note that the file must be in read mode.

**Caution**: Not all formats are suitable for use with this construction. Imagine applying the format *concat("a",anything,"b")* to a very large file. Due to the greedy nature of the scan construct, it will not know when to quit until it has read (into memory!) the entire file (maybe the very last character is a "b").

# Intermediate

### Complex format

```
service {
  session EnterPassword() {
    string s;
    format Alpha = union(range('a', 'z'),
      range('A', 'Z'));
    format Char3x = concat(anychar,anychar,anychar);
    format AtLeast3x = concat(Char3x, anything);
    format HasNonAlpha = complement(star(Alpha));
    format PW = intersection(AtLeast3x, HasNonAlpha);

    html H = <html>
      Enter your password:
      <input type="password" name="p" format="PW">
    </html>;

    show H receive [s = p];
    ...;
  }
}
```

This service has five formats *Alpha*, *Char3x*, *AtLeast3x*, *HasNonAlpha*, and *PW* the goals of which is to define a valid (and restrictive) password. Valid passwords must be at least three characters and contain at least one non-alphanumeric character. The definitions almost speak for themselves. *Alpha* is defined to be any lower or upper case alphanumeric character; *Char3x* to be the concatenation of any three characters; *AtLeast3x* to be at least three characters; *HasNonAlpha* to be any string that has a non alphanumeric character; and finally *PW* to be any string at least three characters long that has a non-alphanumeric character. This format is subsequently bound to a password input field causing the usual incremental validation behavior.

# Advanced

### Match: Strange special cases...

```
service {
  session S() {
    string w = "whatever";
    format Strange = concat([R = anything],
        [S = anything]);

    if (match(w, Strange)[r = R, s = S]) {
      /* Both `r' and `s' are "whatever". */
      ...;
    }
  }
}
```

Due to an overlap in the two formats in the concatenation (that is, the regexp *concat(anything,anything)* is equivalent to *anything*) and the fact that we minimize the deterministic finite-state automata (DFAs) produced from the regular expressions, both *R* and *S* match the string "whatever" in the example. Consequently, both *r* and *s* hold the value "whatever" in the then-branch of the **if**. Also, recordings inside complement constructions may have unpredictable outcomes. [Technically, this is caused by the merging of states in the compositionally produced automata (annotated with alphabet symbols and sets of recording symbols)].

---

# <Database Tutorial>

[ Basics | Intermediate | Advanced ]

<bigwig> is equipped with an internal lightweight database capable of storing all of <bigwig>'s native values. In this way, external shared variables are accessed and manipulated as local ones, presenting the programmer with one uniform concept of data manipulation. A variable is made "shared" (a.k.a. persistent/global/static) by prepending its declaration with the type modifier shared. A **shared** variable is shared among all session threads. Among the native values are composite values of type tuple, relation, and vector.

For the moment, our solution is based on a very general iteration operator called factor, but we are currently integrating an external database with a subset of SQL into <bigwig>. As can be seen in the "SQL macro tutorial", <bigwig>'s syntactic macro abstraction mechanism can be used to clothe our solution as standard SQL queries.

## Basics

### A shared declaration

```
service { // A Page Counter
  session Counter() {
    shared int n;
    /* initially zero by default */

    n++; // increase visible to all session threads
    exit (html) n;
  }
}
```

The variable *n* declared in this example is prefixed with the type modifier shared. Shared variables are shared among all session threads. Thus, when one session thread updates it, subsequent reads in other sessions will get this latest written value. All <bigwig> services have a their own dedicated (internal) database where shared variables are stored.

### Time

```
service {
  session LastAccess() {
    shared time last; // initially `notime'
    time t; // initially `notime'

    t = last;
    last = now();
    if (t==notime) t = now;
    exit (html) t;
  }
}
```

This example just serves to underline that *all* of <bigwig>'s types (except **file** handles) can be made **shared**, even time. The **shared** variable *last* will hold the date and time the session was last accessed and **exit** this value onto the client's browser. Here, the **time** value is output by casting it to an **html** value yielding a default formatting as specified in the type conversion section in the reference manual. Various get*X* functions (where ***X*** is {"Year", "Month", "Day", "Hour", "Minute", "Second", "Weekday"}) exist for referencing the components of a **time** value. These can be used to format **time** values more appropriately.

---

## Vectors

```
service {
  html makeGuestBook(vector string w) {
    int i;
    html H = <html><ul><[guests]></ul></html>;
    html GuestDoc = <html>
      <li><[guest]>
      <[guests]>
    </html>;

    for (i=0; i<|w|; i++) {
      /* |w| is the length of vector w */
      H = H <[guests = GuestDoc <[guest = w[i]]];
    }
    return H;
  }

  session Sign() {
    shared vector string v;
    html SignDoc = <html>
      Please sign the guest book:<br>
      <input type="text" name="guest">
    </html>;
    string s;

    show SignDoc receive [s = guest];
    v = v + vector { s }; // vector constant
    exit makeGuestBook(v); // call-by-value
  }
}
```

This guestbook service has a **shared** vector *v* intended to hold the names of all the people who have signed the guestbook. The *Sign* session initially outputs a document prompting the client for his name. This name is subsequently added to the end of the guestbook list (*v*). The expression "*vector { s }*" is a constant string vector of length one. Finally, a document showing all the names of the people who have signed the guestbook is constructed by a call to a function *makeGuestBook* and **exit**ed to the client. The call to the function *makeGuestBook* causes the vector argument to be copied (call-by-value) and the function will thus operate on this copy (referred to as *w*). The function will iterate through the vector and build a document by plugging the names into document templates producing a document holding the list of the names.

---

**Notes on efficiency!**

```
service {
  shared vector int v;

  session InefficientSum() {
    int i, sum;

    /* highly inefficient!!! */
    for (i=0; i<|v|; i++) { // database lookup (v)
      sum += v[i]; // database lookup (v)
    }
    exit (html) sum;
  }

  session EfficientSum() {
    int i, sum;
    vector int local_v;

    local_v = v; // only one database lookup (v)
    /* highly efficient! */
    for (i=0; i<|local_v|; i++) { // mem. (local_v)
      sum += local_v[i]; // memory lookup (local_v)
    }
    exit (html) sum;
  }
}
```

This service has two sessions both calculating the sum of the integer values in the shared vector *v* (assumed to hold some numbers to be summed). The first session *InefficientSum* shows how not to do this, calculating the sum in a highly inefficient way. The second session *EfficientSum*, however, does the same thing, only efficiently. The *InefficientSum* session iterates through the shared vector using a **for** statement. Notice the reference to *v* (written in red font) in the condition expression and in the statement body of the **for** statement. These two references both cause the shared vector *v* to be read from the database (disk), regardless of whether it has just been read (it could have been modified in between). Consequently, the **for** statement will produce two database lookups for every iteration, yielding a very inefficient calculation. The session *EfficientSum*, however, looks up *v* (in the database) and assigns (by value) this vector to a *local* integer vector variable called *local_v* and performs the sum calculation on this local copy avoiding the many database lookups and yielding a much more efficient calculation. Thus, whenever it is required to iterate through a shared vector structure, it is preferable to ``work on a local copy".

---

### Schemas and tuples

```
service {
  session S() {
    schema Person { // Declare a "Person" schema
      bool is_male;
      int age;
      string name;
    }
    tuple Person p; // Declare p as a Person-tuple
    /* t = tuple {is_male=false, age=0, name=""} */

    p = tuple { is_male=true, age=42, name="John" };
    /* p = tuple {is_male=true,age=42,name="John"} */
    p.age++;
    /* p = tuple {is_male=true,age=43,name="John"} */

    ...;
  }
}
```

This service defines a schema called *Person* which has three components, namely a boolean *is_male*, an integer *age*, and a string *name*. This schema is subsequently used to define a tuple p. Tuples correspond to structs in C and their components initially hold the initial values corresponding to their type. In this example, the variable *p* will initially have *is_male* to **false**, *age* to zero, and *name* to "" (the empty string). The first statement assigns to *p* a constant tuple expression with a schema that is compatible to that of *Person*. The next statement increases the *age* component of *p* by one (from 42 to 43).

---

### Tuples are unordered

```
service {
  session S() {
    schema Person {
      bool is_male;
      int age;
      string name;
    }
    tuple Person p1, p2;

    p1 = tuple { is_male=true, age=42, name="John" };
    p2 = tuple { name="John", is_male=true, age=42 };
    if (p1 == p2) {
      /* They are equal (i.e. exec. proceeds here) */
      ...;
    }
  }
}
```

The purpose of this example is to illustrate that **tuple**s are unordered, meaning that the **if** statement's condition expression will evaluate to **true** as the values held in *p1* and in *p2* are identical.

---

### Tuple manipulation

```
service {
  session S() {
    schema A {
      int n;
      int m;
      float f;
    }
    schema B {
      int n;
      string s;
    }
    schema C {
      int n, m;
      float f;
      string s;
    }
    tuple A a;
    tuple B b;
    tuple C c;

    a = tuple { n=42, f=3.14, m=7 };
    b = tuple { s="foo", n=87 };
    c = a << b; // tuple left-overwrite
    /* c = tuple { n=87, f=3.14, m=7, s="foo"} */
    c.s = "bar";
    /* c = tuple { n=87, f=3.14, m=7, s="bar"} */
    b = c \+ (n, s); // tuple project
    /* b = tuple { n=87, s="bar"} */
    ...;
  }
}
```

This example is not useful as a service but merely meant to illustrate the functionality of the two **tuple** operators "<<" (">>") and "\+" ("\-"). Three schemas *A*, *B*, and *C* are defined and used to declare three **tuple** variables *a*, *b*, and *c*. The first and second statement, assign **tuple** constants to *a* and *b*. The third statement assigns to *c* the ``tuple left overwrite" of *a* and *b*. This operation yields a tuple with a schema that is the union of the schemas of the two arguments. All components with the same names are required to have the same types. The result will contain all the union of the components of the two arguments picking the right one whenever both are present in the two arguments. This operator also has a dual, ">>" that instead picks the component from the left argument in case they are present in both arguments. Thus, *c* will after this operation have 87 as its *n* component. The fourth statement assigns to the *s* component of *c* the constant string "bar". The fifth and final statement assigns to *b*, *c* projected onto the two components; *n* and *s*. A dual tuple operation "\-" exist that instead of naming the components to keep, names the ones to ``throw away".

---

**Vectors of tuples**

```
service {
  session S() {
    int i, age_sum;
    schema Person {
      bool is_male;
      int age;
      string name;
    }
    vector Person v;

    v = vector {
      tuple { is_male=true, age=43, name="John" },
      tuple { is_male=false, age=42, name="Jane" }
    };
    for (i=0; i<|v|; i++) {
      age_sum += v[i].age;
    }
    /* Here, age_sum = 85 */
    ...;
  }
}
```

**Schema**s and **vector**s can also be used to define ``tuple vectors", which are **vector**s with **tuple**s as entries. This service defines a *Person*-**vector** variable *v* and assigns to it a constant **vector** holding two **tuple**s. The **for** statement will subsequently iterate through this vector and calculate the sum of the age components in the vector in *age_sum* (producing 85).

### Relations (no doubles)

```
service {
  session S() {
    int n;
    schema Person {
      bool is_male;
      int age;
      string name;
    }
    relation Person r;

    r = relation {
      tuple { is_male=true,age=43,name="John" },
      tuple { is_male=false,age=42,name="Jane" },
      tuple { is_male=false,age=42,name="Jane" }
    };
    n = |r|; // n is 2 (not 3)
  }
}
```

This service again defines the schema *Person* used in the previous examples. This time, this **schema** is used to define a relation *r* (of *Person*s). Relations differ from vectors in that there is no ordering on the (**tuple**) elements. Thus, the constant relation assigned to *r* in the first statement will immediately ``reduce'' to a relation of ``length'' 2, ignoring the multiplicity of the **tuple** mentioned twice in the constant relation. Consequently, the expression |*r*| evaluates to 2 and not 3. Relations can be seen as a set of **tuples**. The primary operation on relations is called **factor** and is discussed below.

## Intermediate

### Factor (and `#')

```
service {
  session S() {
    int age_sum;
    schema Person {
      bool is_male;
      int age;
      string name;
    }
    relation Person r;

    r = relation {
      tuple { is_male=true, age=38, name="Homer" },
      tuple { is_male=false, age=34, name="Marge" },
      tuple { is_male=true, age=10, name="Bart" }
      tuple { is_male=false, age=8, name="Lisa" }
      tuple { is_male=false, age=1, name="Maggie" }
    };
    factor (r) {
      age_sum += #.age;
    }; // Note: semi-colon required
    /* Here, age_sum is (38+34+10+8+1 =) 91 */
    ...;
  }
}
```

The simplest **factor** expression takes one argument which must be a relation (here *r*). The (statement) body of the **factor** expression will be executed precisely once for each **tuple** in the **relation** given as argument. In each iteration the special (read-only) variable ``#" will be set to the value of the *current* tuple. Thus the above example will calculate the sum of the ages of the persons in the **relation** *r*.

---

**Factor (and return)**

```
service {
  session S() {
    int age_sum;
    schema Person {
      bool is_male;
      int age;
      string name;
    }
    relation Person r, s;

    r = relation {
      tuple { is_male=true, age=38, name="Homer" },
      tuple { is_male=false, age=34, name="Marge" },
      tuple { is_male=true, age=10, name="Bart" }
      tuple { is_male=false, age=8, name="Lisa" }
      tuple { is_male=false, age=1, name="Maggie" }
    };
    s = factor (r) {
      if (#.is_male) {
        return #; // Add current tuple to result.
      }
    };
    /* Here, s contains `homer' and `bart'. */
    ...;
  }
}
```

A **factor** expression evaluates to a **relation**. In the previous example this resulting value was ignored. Here, however, we will assign this value to the **relation** (*Person*) variable *s*. In the (statement) body of a **factor** expression **return** statements are permitted. The value resulting from execution of a **factor** expression will be the union of all the **tuple**s (or **relation**s) **return**ed in its body. Consequently, the value of *s* in the example will be all the tuples for which the *is_male* field is true (that is, `homer' and `bart'). The type of the **tuple**s (or **relation**s) **return**ed are not required to be the same as ``#", but they are required to all be of the same schema which will be the type of the **factor** expression as a whole.

---

### Factor (and identifier arguments)

```
service {
  session S() {
    int n;
    schema Person {
      bool is_male;
      int age;
      string name, hair;
    }
    relation Person r;

    r = relation {
      tuple { is_male=true, age=38,
           name="Homer", hair="none" },
      tuple { is_male=false, age=34,
           name="Marge", hair="blue" },
      tuple { is_male=true, age=10,
           name="Bart", hair="yellow" },
      tuple { is_male=false, age=8,
           name="Lisa", hair="yellow" },
      tuple { is_male=false, age=1,
           name="Maggie", hair="yellow" }
    };
    factor (r; is_male, hair) {
      n++;
    };
    /* Here, n = 4. */
    ...;
  }
}
```

A variant of the **factor** expression takes a comma separated list of identifiers after a semi-colon following the first (**relation**) argument. The relation resulting from the evaluation of the expression argument is projected onto these attributes (which must name attributes in the **relation**al argument) forming a new relation for which any duplicates are removed. The statement will then be executed once per tuple in this relation, setting "#" to the value of the current tuple. Thus, the **factor** expression will iterate through the relation:

```
relation {
  tuple {is_male = true, hair = "none"},
  tuple {is_male = false, hair = "blue"},
  tuple {is_male = true, hair = "yellow"},
  tuple {is_male = false, hair = "yellow"}
}
```

The first three **tuple** are derived from `Homer', `Marge', and `Bart', respectively, whereas the last **tuple** comes from *both* `Lisa' and `Maggie' since they are indistinguishable with respect to gender and hair colour (recall that speaking of the first and fourth tuple really does not make sense since tuples of relations are unordered). The body of the **factor** expression in the example simply increases an integer variable *n* in each iteration of the four **tuple**s, leaving *n* with a final value of 4 after execution of the **factor** expression.

---

## Advanced

---

**Factor (and `@')**

```
service {
  session S() {
    schema Person {
      bool is_male;
      int age;
      string name, hair;
    }
    relation Person r;

    schema AgeName {
      int age;
      string name;
    }
    relation AgeName a;

    r = relation {
      tuple { is_male=true, age=38,
              name="Homer", hair="none" },
      tuple { is_male=false, age=34,
              name="Marge", hair="blue" },
      tuple { is_male=true, age=10,
              name="Bart", hair="yellow" },
      tuple { is_male=false, age=8,
              name="Lisa", hair="yellow" },
      tuple { is_male=false, age=1,
              name="Maggie", hair="yellow" }
    };
    a = factor (r; is_male, hair) {
      if (|@|==2) return @;
    };
    ...;
  }
}
```

Inside the (statement) body of a **factor** expression, the special (read-only) variable ``@'' is available. It will for each **tuple** in the iteration contain a relation with a schema that is the attributes of the relation given to the **factor** expression as argument (here, *is_male*, *age*, *name*, and *hair*), but without the ones names in the identifier list (here *is_male* and *hair*). Thus, in this example, the type of ``@'' is a relation with schema *age* (**int**) and *name* (**string**). The value of ``@'' will in each iteration contain a **relation** with the contributions (with the attributes named in the identifier list projected away) of the **tuple**s of the current **tuple** processed. The **tuples** `Homer', `Marge', and `Bart', all give rise to a ``@'' **relation** of size 1 containing these **tuples**' ages and names. The tuple derived from `Lisa' and `Maggie' will give rise to a ``@'' **relation** of size 2 (which is **return**ed from the **factor** expression). Thus, after the **factor** expression, *a* will contain the following **relation**:

```
relation {
  tuple { age = 8, name = "Lisa" },
  tuple { age = 1, name = "Maggie" }
}
```

---

**Factor (and mutiple relation arguments)**

```
service {
  session S() {
    schema Person {
      int age;
      string name;
    }
    relation Person m, f, all;

    m = relation {
      tuple { age=38, name="Homer" },
      tuple { age=10, name="Bart" }
    };
    f = relation {
      tuple { age=34, name="Marge" },
      tuple { age=8, name="Lisa" },
      tuple { age=1, name="Maggie" }
    };
    all = factor (m, f) {
      return #;
    };
    ...;
  }
}
```

The **factor** expression can also take multiple (**relation**) arguments with the effect that the iteration will be performed on the *intersection* of the relations. The schema of this relation is thus the *intersection* of the two schemas (which in this case is *Person* since both arguments are of schema *Person*). Thus, the variable *all* will after the **factor** expression be:

```
relation {
  tuple { age=38, name="Homer" },
  tuple { age=34, name="Marge" },
  tuple { age=10, name="Bart" },
  tuple { age=8, name="Lisa" },
  tuple { age=1, name="Maggie" }
}
```

---

## Factor (and `@$_n$')

```
service {
  session S() {
    schema Person {
      int age;
      string name;
    }
    relation Person p;

    schema Account {
      int amount;
      string name;
    }
    relation Account a;

    schema PersonAccount {
      int age, amount;
      string name;
    }
    relation Person pa;

    p = relation {
      tuple { age=38, name="Homer" },
      tuple { age=34, name="Marge" },
      tuple { age=10, name="Bart" },
      tuple { age=8, name="Lisa" },
      tuple { age=1, name="Maggie" }
    };
    a = relation {
      tuple { name="Homer", amount=87 },
      tuple { name="Marge", amount=42 },
      tuple { name="Bart", amount=1 },
      tuple { name="Lisa", amount=304 }
    };
    pa = factor (p, a) {
      return cart(relation {#}, cart(@1, @2));
    };
    ...;
  }
}
```

This example exhibits a **factor** expression with two (**relation**) arguments. The iteration will thus be performed on the *intersection* of the relations. The iteration will thus in this example have a schema that is *name* (**string**). The special (read-only) variables ``@1" and ``@2" will for each iteration contain the *rest relations* compared to the first and second arguments of the **factor** expression. Consequently, for the iteration where `#' is equal to `tuple { name="Homer" }`, @1 and @2 will respectively be the relations:

```
relation { tuple { age=38 } }
relation { tuple { amount=87 } }
```

Thus, the variable *pa* will after the **factor** expression be:

```
relation {
  tuple { name="Homer", age=38, amount=87 },
  tuple { name="Marge", age=34, amount=42 },
  tuple { name="Bart", age=10, amount=0 },
  tuple { name="Lisa", age=8, amount=304 }
}
```

# <Concurrency Control Tutorial>

[ Basics | Intermediate | Advanced ]

This tutorial will present how concurrency control is handled in <bigwig>.

Understanding the details of <bigwig>'s concurrency control language **is not essential** for writing <bigwig> services. Most commonly used concurrency control abstractions (such as mutual exclusion regions, reader/writer accessible resources and reader/writer protected shared variables) are provided in the "standard macro library" (std.wigmac) because of <bigwig>'s syntactic macro language. You can get very far using only these macros.

<bigwig> allows the programmer to specify labeled checkpoints (called "wait" statements) in the service code. The order in which session threads pass these checkpoints may be constrained by shared service requirements (called "constraints"). All constraints are written in a logic language which is compiled into a centralized safety controller (process). Each time a session thread wants to pass a checkpoint (a "wait" statement) it asks the safety controller for permission to do so. The controller only grants permission when the entire service is in a state where this is a safe thing to do (that is, when passing a checkpoint does not cause violation of the safety requirements).

Below, you will be shown how to specify constraints, however, most of the time you will simply use the high-level concurrency abstractions provided in the "standard macro library" (std.wigmac).

## Basics

### Wait and constraints

```
service {
  constraint {
    label A; // Declare label `A'.
    /* Safety requirements are specified here.
       Specify when it is safe to pass label
        `A' in service code */
  }

  session S() {
    ...;
    wait A; /* ask controller for permission
      to pass this point. */
    ...;
  }
}
```

This service declares one label (checkpoint) *A* written in a ``constraint { ... }" region. Since there, in this example, are no formulas restricting the *A* label, session threads are at runtime always allowed to pass the wait statement in the session *S*. Each time a session thread passes the wait statement in the session *S*, the controller is asked for permission. Again, since there are no formulas restricting the passing of *A* labels, permission will always (immediately) be given.

---

### Formulas

```
service {
  constraint {
    label A;
    all t: !A(t); /* never grant permission to
      pass `A' */
  }

  session S() {
    ...;
    wait A; // permission is never given!
    ...;
  }
}
```

This service is almost identical to the previous one but with one addition: a safety requirement formula: (*all t: !A(t);*). As one can see the formulas are specified in a logic (monadic first-order logic on strings "M1L-Str" to be exact). The formula states that forall [*all*] possible points [*t*] in time (where some label has been passed) it is not [*!*] the case that [*A(t)*] *A* was passed at time *t*. Consequently, it is never allowed to pass *A* labels and the session *S* will hang indefinitely when it executes the **wait** (*A*) statement. This is not a useful service but it illustrates the basics of defining formulas and how they are used to restrict passing of labels. Usually, a service will have several labels the passing of some will disallow the passing of others until other labels still are passed.

---

## Intermediate

---

### Mutex

```
service {
  constraint {
    label A, B;
    /* Everytime we pass two (different) A's,
       we must have passed a B in between.  */
    all t0: all t2: A(t0) && A(t2) && t0<t2 =>
      is t1: t0<t1 && t1<t2 && B(t1);
  }

  session ExclusiveAccess() {
    wait A;
    /* exclusive access */
    wait B;
  }
}
```

This is a more realistic and useful service example. Two labels *A* and *B* are declared and both restricted by one safety requirement formula. The formula states that in between two different passings of an *A* label, there must be a passing of a *B* label. Consequently, no one is allowed to pass the **wait** (*A*) statement in the session *ExclusiveAccess* when someone has passed the **wait** (*A*) statement and not subsequently passed the **wait** (*B*) statement. Otherwise, the formula would be violated. This yields exclusive access to the statements between the two **wait** statements. The formula yielding mutual exclusion between *A* and *B*, can be defined as a macro so that the programmer does not have to define it every time mutual exclusion is required. In fact, even more powerful abstractions exist that abstracts away from the defining of the labels and the insertion of the **wait** statements around the statements requiring exclusive access. These macros can be found in the standard macro library tutorial: "std.wigmac".

---

**Mutex (using forbid macro)**

```
require <std.wigmac>

service {
  constraint {
    label A, B;
    /* Everytime we pass two (different) A's,
       we must have passed a B in between.  */
    forbid A when is t: A(t) &&
      all tt: t<tt => !B(tt);
  }

  session ExclusiveAccess() {
    wait A;
    /* exclusive access */
    wait B;
  }
}
```

The mutual exclusion formula can also be defined using the forbid-when macros from the standard macro library: "std.wigmac". They enable us to write (a little more directly) that the passing of an *A* is forbidden [*forbid*] when [*when*] there is [*is*] someone that has passed an *A* (at time some point in time [*t*]) and no one has since [*t*<*tt*] has passed a *B*. This formula can be proved semantically equivalent to the one in the previous example.

---

**Mutex (using mutex macro)**

```
require <std.wigmac>

service {
  constraint {
    label A, B;
    mutex(A, B);
  }

  session ExclusiveAccess() {
    wait A;
    /* exclusive access */
    wait B;
  }
}
```

Same example as the previous one, but made by invoking the mutex macro from standard macro library: "std.wigmac".

---

**Region/Exclusive (using region/exclusive macro)**

```
require <std.wigmac>

service {
  region R;

  session ExclusiveAccess() {
    exclusive (R) {
      /* exclusive access */
    }
  }
}
```

Semantically equivalent to the previous example, but made by invoking the region and exclusive macros from standard macro library: "std.wigmac".

---

**A bad idea!**

```
require <std.wigmac>

service {
  html H = ...;
  region R;

  session ExclusiveAccess() {
    exclusive (R) {
      ~~show H;~~ // A bad idea!!!
    }
  }
}
```

One should clearly avoid placing **exit** statements in mutual exclusion regions for the obvious reason, that the exclusive access will never be ``released", causing permanent blocking of the service. Note that the same thing applies for **show** statements, as clients may for some reason decide to not complete the service and walk away in the middle of service execution, leaving the service ``blocked".

This can be partially amended by placing a **wait** (exit-label) statement in the **timeout** of the **show** statement (but the region will be held until the **show** statement times out [See span and SPANDEFAULT] - default is 48 hours).

---

**Initialization example**

```
service {
  shared vector int v;

  constraint {
    label Init, Run;
    /* Everytime we pass `Run', we must have passed
       an `Init' before. */
    all t1: Run(t1) => is t0: t0<t1 && Init(t0);
  }

  session Initialize() {
    v = vector { 42, 87 }; // initialize.
    wait Init; /* Pass `Init' which enables `Run'
      passes. */
  }

  session Ses() {
    wait Run; /* Only pass if someone has passed
       `Init'. */
    ...; // run...
  }
}
```

Often a service has some data the initialization of which is required for the service to behave correctly. This service has a vector *v* which is assumed to hold reasonable data in the session *Ses*. This requirement can easily be specified using <bigwig>'s concurrency control. Two labels *Init* and *Run* are declared and restricted appropriately by a formula. This formula states that forall [*all*] *Run* labels passed (at some point in time [*t1*]), it must be the case that there is [*is*] someone that has before (*t0*<*t1*) that passed (at time *t0*) an *Init* label. Consequently, no one can pass the **wait** (*Run*) statement in the session *Ses*), unless someone has run an *Initialize* session (passed the *Init* label). Although the service has the right intension, it has a very serious problem. Lots of sessions will hang at the **wait** (*Run*) statement (probably very long time) waiting for someone to pass the *Init* label. In some cases this would indeed be interesting (e.g. if the session thread emailed the client when it was eventually allowed to proceed), but in many cases one would probably like to **exit** execution with an appropriate (error) message. We shall see how this is done in the next example:

---

**Wait-branch (timeout)**

```
service {
  shared vector int v;

  constraint {
    label Init, Run;
    all t1: Run(t1) => is t0: t0<t1 && Init(t0);
  }

  session Initialize() {
    v = vector { 42, 87 }; // initialize.
    wait Init; /* Pass `Init' which enables
      `Run' passes. */
  }

  session Ses() {
    wait {
      case Run:
        exit (html) "Service not initialized!";
        break;
      timeout 0:
        /* execution proceeds here if permission
           is not granted to pass checkpoint
           within zero seconds */
        ...; // run...
        break;
    }
  }
}
```

The problem with the previous example that lots of sessions will hang at the **wait** (*Run*) statement, waiting for someone to pass the *Init* label can easily be solved by adding a timeout case in the **wait** statement. The **wait** statement we have seen thus far is in fact a special case of a more general **wait** statement with a **switch**-like syntax. A **wait** statement is allowed to contain a number of **cases** (of labels) plus one **timeout** case. The semantics of the construction is that the session thread waits for permission (from the controller) to pass any one of the listed labels. When permission is given, execution resumes at the statements corresponding to the label passed. If several labels are passable, the controller will *non-deterministically* select one of them. If permission is not given within a certain number of seconds (here zero), the waiting is aborted and execution resumes at the statements specified at the timeout (wait-)branch. Thus, *Ses* sessions started without the service having been initialized, will immediately **exit** with the html (error) message reading *"Service not initialized!"*. Consequently, no services will hang at the **wait** statement in the *Ses* session.

---

# Advanced

---

**All formulas get prefix closed**

```
service {
  constraint {
    label A;
    is t: A(t); // formula gets prefix closed
  }
}
```

All formulas get prefix closed, since it does not make sense to restrict the execution of a service by a formula that is not prefix closed. Hence, the formula *is t: A(t)* has no effect whatsoever, since it will be prefix closed and thus valid on all runs.

---

**Beyond regularity (triggers)**

```
require <std.wigmac>

service {
  shared int n;
  constraint { // Reader/Writer protocol:
    label EnterR, ExitR, EnterW, ExitW;

    mutex(EnterW, ExitW);
    trigger noR when #EnterR == #ExitR;
    allow EnterW when never(EnterR) ||
      (is t: noR(t) &&
        (all tt: t<tt => !EnterR(tt)));
    forbid EnterR when (is t: EnterW(t)) &&
      (all tt: t<tt => !ExitW(tt));
  }

  session S() {
    int x;

    wait EnterR;
    x = n; // read `n'
    wait ExitR;
    ...;
    wait EnterW;
    n = x+1; // write `n'
    wait ExitW;
  }
}
```

Consider the reader/writer protocol bounded by the following constraints:

- At any given time there must be at most one thread writing.
- While there are threads reading there must not be any threads writing.
- While there are threads writing there must not be any threads reading.

Unfortunately, this cannot be expressed in "M2L-Str" since we would have to somehow remember the (*unbounded*!) number of readers at any given moment. The positions at which there are no readers in progress are exacltly those where the number of *EnterR* and *Exit_R* labels occuring before that position are the same. This of course corresponds to the language $\{a^n b^n \mid n >= 0\}$ which is non-regular and therefore cannot be expressed in "M2L-Str". We can of course constrain the problem to any fixed maximum number of readers (*N*) corresponding to $\{a^n b^n \mid N >= n >= 0\}$ which indeed is a regular language (for one thing it is finite) and hence expressible in "M2L-Str".

In order to overcome this hurdle, we have invented a notion of **triggers**. Triggers are constructs that need to be explicitly declared. The declaration...

trigger noR **when** #EnterR == #ExitR;

...will give us a special label (in this case *noR* [no active readers]) that will be passed by the controller once each time the equation becomes true (goes from being false to being true). In this example *noR* will be passed each time the number of *EnterR* and *ExitR* labels become equal.

Again, all of this can transparently be made into a high-level abstraction by <bigwig>'s syntax macro language. Two macros, resource and protected, are available in th standard macro library tutorial: "std.wigmac", implementing the reader/writer protocol.

---

**Protected/Reader/Writer macros**

```
require <std.wigmac>

service {
  protected shared int n;

  session S() {
    int x;

    reader (n) x = n; // read `n'
    ...;
    writer (n) n = x+1; // write `n'
  }
}
```

Semantically equivalent to the previous example, but made by invoking the protected and reader / writer macros from standard macro library: "std.wigmac".

# <Macro Tutorial>

[ Basics | Intermediate | Advanced ]

All macros in <bigwig> must be placed in *packages* (that is, *files*) and **require**d by the service in order to be available for usage in the service program.

## Basics

### A very simple macro: Pi

```
                        ---pi.wigmac---

syntax <floatconst> pi ::= {
  3.1415926
}
```

```
require "pi.wigmac"

service {
  session S() {
    float f;

    f = pi;
    ...;
  }
}
```

One of the simplest macros one could write, would be a macro that does not take any arguments as is the case for the macro pi in the example. When declared it will appear to the programmer as if the *floatconst* syntactic category had been extended with a production pi. This is different from a lexical macro in that the macro invocation of pi is only allowed in places where a *floatconst* would be. Also, the body of the macro is parsed and thus syntax checked to see if it complies with the declared return type (here *floatconst*) at definition time (not expansion time). This ensures that no parse errors are produced as a consequence of invoking the macro, regardless of invocation context.

### A macro with an argument: Maybe

```
                    ---maybe.wigmac---

syntax <stm> maybe <stm S> ::= {
  if(random(2)==1) <S>
}
```

```
require "maybe.wigmac"

service {
  session S() {
    html H = ...;

    maybe show H;
    ...;
  }
}
```

This example defines a new construct, maybe, that takes an argument, namely a statement and executes it with 50% probability. The argument taken (*S*) refer to a syntax tree (of kind statement) produced from parsing a statement and is in the body of the macro referred to as a normal identifier in angled brackets (<S>).

---

### A macro definition with token separators: Plus

```
                    ---plus.wigmac---

syntax <regexp> plus ( <regexp R> ) ::= {
  concat(<R>, star(<R>))
}
```

```
require "plus.wigmac"

service {
  session S() {
    format Digit = range('0', '9');
    format Number = plus(Digit);

    ...;
  }
}
```

In the <bigwig> grammar, one can see that there is something called star for Kleene's star on regular languages, signaling zero-or-more. However, there is nothing called "plus" for one-or-more. Such a construct could easily be defined by a macro. This is a nice example of a macro that uses token separators to enforce a particular syntax. The macro definition contains two *tokens*, namely the two parentheses. The compiler will thus automatically enforce that invocations of the macro plus contain the two parentheses in the sense that it would be a syntactic error to omit them. In this way the macro author can tailor the macros to have the desired look-and-feel.

**Caution**: this extreme flexibility could be abused to write macros that expected horrific syntax with, for instance, unbalanced parentheses of varying kinds. So the macro programmer should take some care when designing a macro's invocation syntax.

---

## A macro with an identifier delimiter: Repeat-until

```
                  ---repeat_until.wigmac---

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    <S>
    while (!<E>) <S>
  }
}
```

```
require "repeat_until.wigmac"

service {
  session S() {
    int x, y, z;

    ...;
    repeat {
      x = x * y;
      y--;
    } until (y==0);
    ...;
  }
}
```

This macro, repeat-**until** will take two arguments, *S* and *E*, delimited not only by tokens but also by an identifier "until". The repeat-**until** construct is *transparent* in the sense that it appears to the programmer as if it really was in the language.

**Caution**: The statement argument *S* is used twice in the body of the macro and because arguments are copied (for good reason), the repeat-**until** macro may cause an explosion in the size of the parse tree produced when the construct is nested. We shall see later how to define the repeat-**until** macro without potential parse tree explosion.

---

## A macro in terms of another: Forever

```
                    ---forever.wigmac---

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    <S>
    while (!<E>) <S>
  }
}

syntax <stm> forever <stm S> ::= {
  repeat <S> until (false);
}
```

```
require "forever.wigmac"

service {
  session S() {
    html H = ...;

    forever show H;
  }
}
```

A macro can easily be defined in terms of another as is the case in this example, where we have defined a new macro forever in terms of repeat-**until**. The repeat-**until** macro will only be expanded once, namely when the forever macro is defined (i.e. parsed).

---

**2pass scope rules**

```
                    ---forever.wigmac---

syntax <stm> forever <stm S> ::= {
  repeat <S> until (false);
}

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    <S>
    while (!<E>) <S>
  }
}
```

```
require "forever.wigmac"

service {
  session S() {
    html H = ...;

    forever show H;
  }
}
```

<bigwig> macro definitions have two-pass scope rules, which means that a macro is available in a package even before its lexical point of definition. This service is semantically equivalent to the previous one.

---

**Recursion not allowed!**

```
                    ---illegal.wigmac---

syntax <stm> Rec ::= {
  ...~~Rec~~...;
}
```

```
require "illegal.wigmac"

service {
  ...
}
```

Without a compile-time language for ``breaking the recursion'', recursion can only yield infinite program fragments. Therefore, macro recursion (and mutual recursion) is not permittet in <bigwig>. However, as we shall see later, <bigwig> supports metamorphic macros which enable recursive definitions that yield finite expansions.

---

**Stacking packages**

```
                  ---repeat_until.wigmac---

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    <S>
    while (!<E>) <S>
  }
}
```

```
                    ---forever.wigmac---

require "repeat_until.wigmac"

syntax <stm> forever <stm S> ::= {
  repeat <S> until (false);
}
```

```
require "forever.wigmac"

/* Only the `forever' macro is available here! */
service {
  session S() {
    html H = ...;

    forever show H;
  }
}
```

This service is again semantically equivalent to the two previous ones. Here, we have split the package into two packages, one for each macro. The *forever* package **require**s the *repeat* package in order to use it. Since the *repeat* package is **require**d (as opposed to **extend**ed), it is local to the *forever* package and will not be exported beyond it (to the service). Thus, only the *forever* macro is available in the service.

## Extend

```
                  ---repeat_until.wigmac---

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    <S>
    while (!<E>) <S>
  }
}
```

```
                    ---forever.wigmac---

extend "repeat_until.wigmac"

syntax <stm> forever <stm S> ::= {
  repeat <S> until (false);
}
```

```
require "forever.wigmac"

/* Both macros are available here! */
service {
  session S() {
    html H = ...;

    repeat show H; until (1==2);
  }
}
```

This service is slightly different than the previous one in that the *forever* package **extend**s the *repeat* package. This means that both macros are exported from the *forever* package into the service.

---

# Intermediate

---

**Specificity (split: ``end'' vs. terminal)**

```
                            ---split.wigmac---

syntax <stm> si (<exp E>) <stm S1> ::= {
  if (<E>) <S1>
}

syntax <stm> si (<exp E>) <stm S1> sinon <stm S2>
    ::= {
  if (<E>) <S1> else <S2>
}
```

---

```
require "split.wigmac"

service {
  session S() {
    int x,y;

    ...;
    si (f(x)>0) si (f(y)<0) x++; sinon y--;
    ...;
  }
}
```

In <bigwig> several macros may have the same name. However, macros with the same name must all have the same nonterminal return type. Also, their headers (invocation syntax design) are not allowed to be exactly the same which means that at some point the headers are said to *split*. In the example the two **si** macros have same headers up until after the statement argument (S1); after which the first macro ``ends" and the second has a (terminal symbol) "sinon" identifier. Invocation parsing always selects the most *specific* definition for an invocation. In the example, the first "si" will commence macro parsing, the second "si" will commence another instance of macro parsing and will after the "x++;" statement be left with a choice of whether to take the first definition (stop) or the second (match the "sinon"). The macros are greedy, to the second definition will be taken; correctly solving the *dangling-sinon* problem.

**In detail:** There are three kinds of header elements; terminals, nonterminals, and ``end". Invocation parsing is conducted in *challenge rounds*, header element after header element. Whenever a macro invocation is parsed, the current token on the input stream is matched against all (live) macro definitions to see if they are compatible. Those macro definitions that are not are immediately eliminated (set to non-live). Between the compatible macro definitions, the ones (there may be more than one) that are the most *specific* are kept, the others are eliminated. The following specificity relation is used:

```
terminal < nonterminal < ``end''
```

## Specificity (split: terminal vs. nonterminal)

```
                    ---split.wigmac---

syntax <stm> pay 1 dollar ; ::= {
  ...1...;
}

syntax <stm> pay <exp E> dollars ; ::= {
  ...<E>...;
}
```

```
require "split.wigmac"

service {
  session S() {
    ...;
    pay 1 dollar;  // 1st definition.
    pay 7 dollars; // 2nd definition.

    ...;
  }
}
```

This example again illustrates the specificity splitting. The "1" in the first macro invocation "pay 1 dollar;" matches both definitions, since "1" is in the set of first-tokens of expressions. However, due to the specificity relation, the terminal is favored over the nonterminal; meaning that the first definition is chosen.

## Pretty printer directives

```
                    ---split.wigmac---

syntax <stm> si (<exp E>) <stm S1> ::= {
  if (<E>) <S1>
}

syntax <stm> si (<exp E>) <stm S1> \n sinon <stm S2>
    ::= {
  if (<E>) <S1> else <S2>
}
```

```
require "split.wigmac"

service {
  session S() {
    int x,y;

    ...;
    si (f(x)>0) si (f(y)<0) x++; sinon y--;
    ...;
  }
}
```

The <bigwig> pretty printer is able to ``unparse" the source code with or without expanding macros. When unparsing without macro expansion, the macro programmer is allowed to instruct the pretty printer how to pretty print a macro invocation. That is, when to insert newlines, whitespaces, even how to indent a macro invocation. The characters `\n` (newline), `\+` (indent), and `\-` (unindent) are available for this. In the example, the "si-sinon" macro is instructed to print a newline character before the "sinon". Also, whitespaces are significant in the definition headers. Note that there are no spaces between the expression argument (E) and the parentheses. This means that the expression will be printed in parentheses without spaces.

---

**Repeat-until: Alpha-conversion**

```
                    ---alpha.wigmac---

syntax <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    bool first = true; // first is alpha-converted!

    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

```
require "alpha.wigmac"

service {
  session S() {
    int x, y, z;

    ...;
    repeat {
      x = x * y;
      y--;
    } until (y==0);
    ...;
  }
}
```

As promised we define a repeat-**until** macro that does not cause parse tree explosion. We exploit runtime behaviour of <bigwig> (the host language) to give our macro the right semantics. We introduce a new **bool** variable *first* to remember (at runtime) whether or not we have executed the statement *S*. Note that as in **C**, the boolean-or operator "||" is *lazy* and since *first* is initially **true**, *E* never gets evaluated the ``first time around''. One could suspect that invocations of such a macro could yield identifier clashes if the statement contained a reference to an identifier *first* that was defined outside the invocation. However, with the <bigwig> macros this is not the case. Any non-argument identifier in a macro is automatically alpha-converted (renamed) to avoid accidental name-clashes. This hygienic property allows us to safely define macros such as the one above.

---

**Alpha-conversion suppression: ` (operator)**

```
                    ---declare_i.wigmac---

syntax <decl> declare_i ; ::= {
  int `i = 42; // Caution!
}
```

```
require "declare_i.wigmac"

service {
  session S() {
    declare_i;

    i = 87;
    ...;
  }
}
```

We have as a feature introduced an alpha-conversion suppression operator "`" (backping). So in this example, the code produced by the invocation of the macro declare_i declares an identifier named exactly *i* which is the one subsequently assigned 87.

**Caution**: This operator should be used with caution, as macros defined using it become highly context sensitive!

# Advanced

This section of the macro tutorial will present the concept of *metamorphisms*. A *metamorphism* is a user defined (*meta*-grammar) nonterminal along with a rule specifying how the new (macro) syntax is *morphed* into host language (`<bigwig>`) syntax. As we shall see, metamorphisms can be used to define macros accepting almost arbitrary syntax with a varying number of arguments.

We will commence by gently introducing metamorphisms through a couple of *toy examples* after which we will look at some "real" examples.

**Metamorphisms: Abbreviation**

```
                         ---si.wigmac---

metamorph <exp> until_part --> until (<exp Cond>) ::=
{
  <Cond>
}

syntax <stm> repeat <stm S> <until_part: exp E> ; ::=
{
  {
    <S>
    while (!<E>) <S>
  }
}
```

```
require "si.wigmac"

service {
  session S() {
    int x, y, z;

    ...;
    repeat {
      x = x * y;
      y--;
    } until (y==0);
    ...;
  }
}
```

A simple use of metamorphisms is to abbreviate definitions. In this toy example we have written the previous "repeat-until" macro, but where we have cut the definition in two parts by introducing a new nonterminal *until_part*. When a "repeat" macro invocation is being parsed and the parser comes to the metamorph argument...

```
<until_part: exp E>
```

...the parser will parse a (user defined) *until_part* which will yield an expression that will be called (E). Hereafter, parsing of the repeat macro is resumed, which will parse the terminal "`;`". This example is not very useful in the sense that it could easily have been defined as a (normal) macro, but it serves to illustrate metamorphisms.

---

**Metamorphisms: Splitting**

```
                        ---si.wigmac---

metamorph <stm> opt_else --> ::= {
  /* empty */;
}

metamorph <stm> opt_else --> sinon <stm S> ::= {
  <S>
}

syntax <stm> si (<exp E>) <stm S1>
    <opt_else: stm S2> ::= {
  if (<E>) <S1> else <S2>
}
```

```
require "si.wigmac"

service {
  session S() {
    int x,y;

    ...;
    si (f(x)>0) si (f(y)<0) x++; sinon y--;
    ...;
  }
}
```

The same splitting rules we saw for the (normal) macros also apply to metamorphisms. The specificity relation is extended in the following way:

```
terminal < nonterminal < metamorph < ``end''
```

---

**Metamorphisms: Lists**

```
                        ---xlist.wigmac---

metamorph <stm> xlist --> ::= {
  /* empty */;
}

metamorph <stm> xlist --> X <xlist: stm S> ::= {
  <S>
}

syntax <stm> Xlist <xlist: stm S> ; ::= {
  <S>
}
```

```
require "xlist.wigmac"

service {
  session S() {
    Xlist X X X X;
  }
}
```

This example shows how lists can easily be defined with metamorphisms. The macro "Xlist" takes a metamorph argument *xlist* that can do one of two things; stop or parse an "X" followed by an ``*xlist*". The result is that the macro will parse "Xlist" followed by an arbitrary number of "X"'es and transform this into the empty statement.

---

**Metamorphisms: No Left Recursion!**

```
                        ---illegal.wigmac---

metamorph <stm> xlist --> ::= {
  /* empty */;
}

metamorph <stm> xlist --> ~~<xlist: stm S>~~ X ::= {
  <S>
}

syntax <stm> Xlist <xlist: stm S> ; ::= {
  <S>
}
```

```
require "illegal.wigmac"

service {
  session S() {
    Xlist X X X X;
  }
}
```

This is an example of an illegal service. Since the parser is a modified LL(1) top-down parser, left recursion would cause it to loop. Consequently, left recursion is intercepted and rejected by the parser. One should instead make the metamorphisms right recursive as in the previous example.

---

**Metamorphisms: Productivity!**

```
                    ---illegal.wigmac---

~~metamorph <stm> xlist --> X <xlist: stm S>~~ ::= {
  <S>
}

syntax <stm> Xlist <xlist: stm S> ; ::= {
  <S>
}
```

```
require "illegal.wigmac"

service {
  session S() {
    Xlist X X X X ...
  }
}
```

This is another example of an illegal macro. As a sanity check, the <bigwig> parser will check that all metamorphisms derive something (finite). The *xlist* nonterminal in this example cannot derive something finite and is thus rejected.

---

**Metamorphisms: Palindrome**

```
                   ---palindrome.wigmac---

metamorph <stm> p --> C ::= {
  /* empty */;
}

metamorph <stm> p --> A <p: stm S> A ::= {
  <S>
}

metamorph <stm> p --> B <p: stm S> B ::= {
  <S>
}
```

```
syntax <stm> palindrome ( <p: stm S> ) ; ::= {
  <S>
}
```

---

```
require "palindrome.wigmac"

service {
  session S() {
    palindrome ( A B A C A B A );
  }
}
```

So far we have seen how metamorphisms could be used to parse *list structures*. This (toy) example shows how also *tree structures* can be parsed. The macro "palindrome" accepts palindromes over the alphabet {"A", "B"} with a "C" in the middle.

---

**Metamorphisms: Enum**

```
                       ---enum.wigmac---

syntax <decl_list> enum { <id I>
    <enum_list: decl_list EL> } ; ::= {
  int e;
  const int <I> = e++;
  <EL>
}

metamorph <decl_list> enum_list --> ::= {
}

metamorph <decl_list> enum_list --> , <id I>
    <enum_list: decl_list EL> ::= {
  const int <I> = e++;
  <EL>
}
```

---

```
require "enum.wigmac"

service {
  session S() {
    int n;
    enum { ZERO, ONE, TWO, THREE };

    n = ONE+TWO*THREE;
  }
}
```

This real example shows how **enum**s can easily be added to the `<bigwig>` language. The "enum" macro produces a list of declarations using an integer variable for *e* generating the enumeration values. This *e* will be alpha converted to the same fresh identifier in the macro and the two metamorphisms.

---

## Metamorphisms: Switch

```
                     ---Switch.wigmac---

syntax <stm> Switch { \+\n <swbody: stm S> \-\n } ::=
{
  {
    typeof(<E>) x = <E>;
    <S>
  }
}

metamorph <stm> swbody --> Case <exp E> :
    <stm_list SL> Break ; <swbody: stm S> ::= {
  if (x==<E>) { <SL> } else <S>
}

metamorph <stm> swbody --> Case <exp E> :
    <stm_list SL> Break ; ::= {
  if (x==<E>) { <SL> }
}
```

```
require "Switch.wigmac"

service {
  session S() {
    int x, y, dot_com, dot_org;
    string s;

    ...;
    Switch (s) {
      Case ".com":
        dot_com++;
        f(x);
        Break;
      Case ".org":
        dot_org++;
        g(y);
        Break;
    }
  }
}
```

This example shows how to make the "Switch" statement with a capital S (because `<bigwig>` already has a **switch** statement).

---

See also: The Standard Macro Library Tutorial

---

# <Standard Macro Library Tutorial>

**Note**: This tutorial assumes basic knowledge on:

- Concurrency Control

### allow-when (in terms of restrict-by)

```
syntax <formula> allow <id L> when <formula F> ::= {
  all t: <L>(t) => restrict <F> by t
}
```

```
require <std.wigmac>

service {
  constraint {
    label Run, Init;
    allow Run when is t: Init(t);
  }
}
```

### forbid-when (in terms of allow-when)

```
syntax <formula> forbid <id L> when <formula F> ::= {
  allow <L> when !<F>
}
```

```
require <std.wigmac>

service {
  constraint {
    label Run, Init;
    forbid Run when all t: !Init(t);
  }
}
```

### mutex (in terms of forbid-when)

```
syntax <formula> mutex ( <id A> , <id B> ) ::= {
  forbid <A> when is t: <A>(t) &&
    (all tt: t<tt => !<B>(tt))
}
```

```
require <std.wigmac>

service {
  constraint {
    label Enter, Exit;
    mutex(Enter, Exit);
  }
}
```

---

**region (in terms of mutex)**

```
syntax <toplevel> region <id R> ; ::= {
  constraint {
    label <R>~A, <R>~B;
    mutex(<R>~A, <R>~B);
  }
}

syntax <stm> exclusive ( <id R> ) <stm S> ::= {
  {
    wait <R>~A;
    <S>
    wait <R>~B;
  }
}
```

```
require <std.wigmac>

service {
  session S() {
    shared int x, y;
    region reg;

    ...;
    exclusive (reg) {
      int temp = x; // swap x and y (atomically)

      x = y;
      y = temp;
    }
    ...;
  }
}
```

## resource (in terms of region)

```
syntax <toplevel> resource <id R> ::= {
  region <R>;

  constraint {
    label <R>~enterR, <R>~exitR, <R>~P;
    trigger <R>~RC when #<R>~enterR == #<R>~exitR;
    trigger <R>~PC when #<R>~P == #<R>~B;
    allow <R>~enterR when never(<R>~P) ||
      (is t: <R>~PC(t) &&
        (all tt: t<tt => !<R>~P(tt)));
    allow <R>~A when never(<R>~enterR) ||
      (is t: <R>~RC(t) &&
        (all tt: t<tt => !<R>~enterR(tt)));
  }
}

syntax <stm> reader ( <id R> ) <stm S> ::= {
  {
    wait <R>~enterR;
    <S>
    wait <R>~exitR;
  }
}

syntax <stm> writer ( <id R> ) <stm S> ::= {
  {
    wait <R>~P;
    exclusive (<R>) <S>
  }
}
```

---

```
require <std.wigmac>

service {
  session S() {
    int n;
    shared int x;
    resource res;

    ...;
    reader (res) n = x;
    ...;
    writer (res) {
      n = x + n*3;
      x = x * 2;
    }
    ...;
  }
}
```

---

**protected (in terms of resource)**

```
syntax <toplevel_list> protected <type T> <id V> ;
    ::= {
  <T> <V>;

  resource <V>;
}
```

---

```
require <std.wigmac>

service {
  session S() {
    protected shared int n;

    ...;
  }
}
```

# <SQL Macro Library Tutorial>

**Note**: This tutorial assumes basic knowledge on:

- Database

## join

```
syntax <exp> join (<exp R1>, <exp R2>) ::= {
  factor(<R1>, <R2>) {
    return cart(relation{ # }, cart(@1, @2));
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    rs = join (r,s);
  }
}
```

## Union

```
syntax <exp> Union (<exp R1>, <exp R2>) ::= {
  factor(<R1>, <R2>) {
    return #;
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    rs = Union (r,s);
  }
}
```

## Select

```
syntax <exp> Select * from <exp R> ::= {
  factor(<R>) {
    return #;
  }
}
```

---

```
require <sql.wigmac>

service {
  session S() {
    ...;
    s = Select * from r;
  }
}
```

---

## Select

```
syntax <exp> Select * from <exp R> where <exp C>
    ::= {
  factor(<R>) {
    if (<C>) {
      return #;
    }
  }
}
```

---

```
require <sql.wigmac>

service {
  session S() {
    ...;
    s = Select * from r where id!=0;
  }
}
```

---

## Delete

```
syntax <exp> Delete from <exp R> where <exp C>
    ::= {
  <R> = factor(<R>) {
    if (!<C>) {
      return #;
    }
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    Delete from r where id==0;
  }
}
```

## project

```
syntax <exp> project <exp R> on (<id_list IL>)
    ::= {
  factor(<R>) {
    return # \+ (<IL>);
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    s = project r on (id, name);
  }
}
```

## Update

```
syntax <exp> Update <exp R1> set <exp New> where
    <exp C> ::= {
  <R1> = factor(<R1>) {
    if (<C>) {
      return <New>;
    } else {
      return #;
    }
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    Update r set t where id==0;
  }
}
```

## Update

```
syntax <exp> Update (<exp R1> with <exp R2> using
    <id_list IL> ::= {
  factor(<R1>, <R2>; <IL>) {
    if (|@2|==0) return cart(@1,relation{#});
    else return cart(@2,relation{#});
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    rs = Update r with s using id, name;
  }
}
```

## rename

```
syntax <exp> rename in <exp R> from <id A> to <id B>
    ::= {
  factor(<R>) {
    return (# \- (<A>)) << tuple { <B> = #.<A> };
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    s = rename in r from id to key;
  }
}
```

---

**difference**

```
syntax <exp> difference (<exp R1>, <exp R2>) ::= {
  factor(<R1>, <R2>) {
    if (|@2| == 0) {
      return #;
    }
  }
}
```

```
require <sql.wigmac>

service {
  session S() {
    ...;
    rs = difference (r,s);
  }
}
```

---

**aggregate**

```
syntax <exp> aggregate (<exp R>, <stm S>) ::= {
  factor(<R>) {
    <S>
  }
}
```

---

```
require <sql.wigmac>

service {
  session S() {
    ...;
    rs = aggregate (r,s);
  }
}
```