Basic Research in Computer Science

BRICS NS-00-4 C. Brabrand: <bigwig> Version 1.3 — Reference Manual

<bigwig> Version 1.3 Reference Manual

Claus Brabrand

BRICS Notes Series NS-00-4

ISSN 0909-3206 September 2000

Copyright © 2000, Claus Brabrand.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent BRICS Notes Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/00/4/`

<bigwig> Version 1.3
Reference Manual

Claus Brabrand
brabrand@brics.dk

September 2000

<Reference Manual>

This reference manual concisely describes the whole <bigwig> language. We advise new <bigwig> programmers to start by reading the tutorial and studying the examples.

1. Keyword Index
Page 2

2. Lexical Structure
Page 3

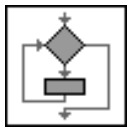
3. Syntax
Page 5

4. Operators
Page 13

5. Scope Rules
Page 15

6. Types
Page 16

7. Core Language
Page 20



8. Control Structures
Page 22

9. Files
Page 25

10. Macros
Page 27

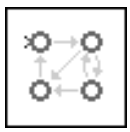
11. Form Input
Page 30

12. Formats
Page 33

13. Dynamic Documents
Page 37

14. Database
Page 44

15. Security
Page 47



16. Concurrency Control
Page 48

17. Web Specifics
Page 52

18. Time
Page 55

<Keyword Index>

[A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z]

This section contains a list of all keywords in the <bigwig> language and macros in the standard macro package, "std.wigmac". Black names are keywords and *blue* names are the macros defined in the standard macro package and are thus only active in a service program if this package is required.

A: **all** *allow* **anychar**
B: **bool break by**
C: **cart case char** *charcomplement* **close complement concat const constraint continue**
D: **default difftime dir** *do*
E: **else eof** *exclusive* **exit extend**
F: **factor false file fileerror fix flash float for** *forbid* *forever* **format**
G: **get getcookie getenv getYear getMonth getDay getHour getMinute getSecond getWeekday**
H: **htaccess html**
I: **if ignoreformats** #include **int intersection is**
J: –
K: –
L: **label** *loop*
M: **macro match** *maybe* md5 **metamorph** *mutex*
N: *never* **notime now**
O: **open** *optional*
P: *pi* *plus* **post print println** *protected*
Q: –
R: **random range rawhtml** *reader* **receive refresh regexp** *region* **relation relax** *repeat* **require** *resource* **restrict return**
S: **scan scanln schema selective** *sendemail* **sendmail service session setcookie** *setYear* *setMonth* *setDay* *setHour* *setMinute* *setSecond* **shared show singular sort span ssl star string switch syntax system**
T: **time timeout track trigger true tuple typeof**
U: **union url userid**
V: **vector void**
W: **wait when while** *writer*
X: –
Y: –
Z: –

<Lexical Structure>

[Tokens | Case-sensitivity | Comments | Lexical Inclusion | Macros]

Tokens

The table below contains an explanation of the seven token kinds, INT, FLOAT, CHAR, STRING, IDENTIFIER, DEFATTR, and WHATEVER.

Token	Regexp (Yacc)	Examples
INT	`[0-9]+`	`0, 7, 42, 087`
FLOAT	`[0-9]* \. [0-9]+`	`0.0, .99, 3.14, 01.10`
CHAR	`'[^\"'\n\t]'`	`'c', ' ', 'ø', '\n', '\t'`
STRING	`\"((\\\")|([^\"\n\t]))*\"`	`"", "c", "foo", "hello !\n"`
IDENTIFIER	`[a-zA-Z]([a-zA-Z0-9]|(_[a-zA-Z0-9]))*`	`x, x87, X_1, name, Amount`
DEFATTR	`[^ \n\t&<>\"=\[\]]+`	`x, 87, -8a, xyz, år, foo'`
WHATEVER	`[^<>\n]*`	`basically, any!, ' $line@#`

Case-sensitivity

<bigwig> is *case-sensitive* meaning that it distinguishes between upper and lower-case characters. Thus the identifiers "amount", "Amount", and "AMOUNT" are different. However, within html-documents (excepting gap names), <bigwig> is *case-insensitive*. For instance, **<textarea>**, **<Textarea>**, and **<TEXTAREA>** mean the same.

Comments

There are two way of forming comments in <bigwig>. As in C or Java, comments are written within `/*` (slash-asterisk) and `*/` (asterisk-slash). However, unlike the two languages, <bigwig> supports arbitrary nesting of comments. Also, comments can be formed using `//` (double-slash), with the effect that the rest of the line is taken as a comment. The two ways of forming comments are respectively referred to as *region comments* and *line comments*. In HTML mode the usual `<!-- ... -->` comments apply, meaning that anything within is ignored by <bigwig>. Note, however, that such comments *are* generated in the document output.

Lexical Inclusion

As in C, lexical file inclusion is performed by the directive "`#include`". This directive takes one

argument, namely a URL in quotes. The URL can be either a file or a hyper-reference (beginning with "`http://`"). If a file is specified, <bigwig> will fetch the file from the file-system, as in C. If a hyper-reference is specified, the URL-file will be fetched from the Internet (using W3C's Libwww) at compile-time. The URL can also be supplied in angled brackets, in which case the URL is prepended with a string (defaulting to "`http://www.brics.dk/bigwig/macro/`"). This is typically used for including macro packages from "<bigwig>-central", but the prepend-string can be altered in <bigwig>'s configuration file "`.bigwig`".

Macros

<bigwig> provides no *lexical* macro concept, but has a much more powerful *syntactic* macro concept.

<Syntax>

The BNF definition of the <bigwig> syntax:

bigwig	: require_list service
require_list	: require *
require	: **require** <URL> | **require** stringconst
service	: mode_list **service** { toplevel_list }
toplevel_list	: toplevel *
toplevel	: mode_list session | decl_list | constraint | format | schema
session	: **session** id (argument_list) { toplevel_list stm_list }
mode_list	: mode *
mode	: **ssl** | **singular** | **selective** (exp_list) | **span** (exp) | **refresh** (exp) | **htaccess** (exp)
html	: <**html**> htmlbody_list </**html**> | <**html**> htmlbody_list </**html**> @ stringconst
htmlbody_list	: htmlbody *
htmlbody	: < defattr attribute_list > | </ defattr > | <**bigwig**> | <**title**> htmlbody </**title**> | <**body**> htmlbody </**body**> | WHATEVER | <!-- ... --> | < [ ] >

| < [id] >
| < [(exp)] >
| < [compound_stm] >
| <**input** attribute_list >
| <**textarea** attribute_list > htmlbody_list </**textarea**>
| <**select** attribute_list > htmlbody_list </**select**>
| <**tuple** attribute_list > htmlbody_list </**tuple**>
| <**continue** attribute_list > htmlbody_list </**continue**>
| <**applet** attribute_list > htmlbody_list </**applet**>
| <**param** attribute_list >
| <**result** attribute_list >

attribute_list : attribute*

attribute : mode_list
| **name** = id
| **format** = id
| attr
| attr = attr

attr : defattr
| [ ]
| [id]
| [(exp)]
| [compound_stm]

defattr : DEFATTR
| STRINGCONST

---

constraint : **constraint** { constraintbody_list }

constraintbody_list : constraintbody+

constraintbody : label
| trigger
| formula **;**

label : **label** id_list **;**

trigger : **trigger** id **when** trigexp == trigexp **;**

trigexp : trigexp + trigexp
| trigexp − trigexp
| + trigexp
| − trigexp
| intconst
| # id

formula : **true**
| **false**

| **restrict** formula **by** id
| **is** id **:** formula
| **all** id **:** formula
| formula **&&** formula
| formula **||** formula
| formula => formula
| formula <=> formula
| **!** formula
| id == id
| id **!=** id
| id < id
| id > id
| id <= id
| id >= id
| id **(** id **)**
| **(** formula **)**

---

format : **format** id = regexp **;**

regexp_list : regexp⊕

regexp : id
| stringconst
| **anychar**
| **star** **(** regexp **)**
| **concat** **(** regexp_list **)**
| **union** **(** regexp_list **)**
| **intersection** **(** regexp_list **)**
| **complement** **(** regexp **)**
| **range** **(** stringconst **,** stringconst **)**
| **fix** **(** intconst **,** intconst **)**
| **relax** **(** intconst **,** intconst **)**
| **regexp** **(** stringconst **)**
| **[** id = regexp **]**

---

schema : **schema** id **{** field_list **}**

field_list : field*

field : type id_list **;**

---

decl_list : decl*

decl : type id_list **;**
| type id_list = exp **;**
| type id_list **(** argument_list **)** compound_stm

| type id_list (argument_list) ;

argument_list : argument⊛

argument : type id

---

type : modifier_list **void**
| modifier_list **bool**
| modifier_list **int**
| modifier_list **float**
| modifier_list **char**
| modifier_list **string**
| modifier_list **time**
| modifier_list **file**
| modifier_list **tuple** id
| modifier_list **relation** id
| modifier_list **vector** id
| modifier_list **vector** type
| modifier_list **html**
| **typeof** (exp)

modifier_list : modifier*

modifier : **const**
| **shared**

---

stm_list : stm*

stm : */* empty */* **;**
| exp **;**
| **exit ;**
| **exit** mode_list exp **;**
| **flash** mode_list exp **;**
| **show** mode_list exp **;**
| **show** mode_list exp **timeout** stm
| **show** mode_list exp **receive** [getinput_list] **;**
| **show** mode_list exp **receive** [getinput_list] **timeout** stm
| **wait** id **;**
| **wait** { waitbranch_list }
| **return ;**
| **return** exp **;**
| **if** (exp) stm
| **if** (exp) stm **else** stm
| **while** (exp) stm
| **for** (exp? **;** exp? **;** exp?) stm

| **switch** (exp) { switchbranch_list }
| **sendmail** (exp , exp)
| compound_stm

compound_stm : { decl_list stm_list }

waitbranch_list : waitbranch⊕

waitbranch : **case** id : stm_list **break ;**
| **timeout** exp : stm_list **break ;**

switchbranch_list : switchbranch⊕

switchbranch : **case** exp : stm_list **break ;**
| **default** : stm_list **break ;**

getinput_list : getinput⊛

getinput : exp = id

---

exp_list : exp⊛

exp : #
| @
| @ intconst
| **dir**
| **url**
| **url** id (exp_list)
| **userid**
| **time** (exp , exp , exp)
| **time** (exp , exp , exp , exp , exp , exp)
| **now**
| **notime**
| **difftime** (exp , exp)
| boolconst
| intconst
| floatconst
| charconst
| stringconst
| **tuple** { tupleexp_list }
| **relation** { exp_list }
| **vector** { exp_list }
| html
| id
| id (exp_list)
| (exp)
| + exp
| − exp

| exp = exp
| exp += exp
| exp −= exp
| exp *= exp
| exp /= exp
| exp %= exp
| exp =< [plug_list]
| ++ exp
| exp ++
| −− exp
| exp −−
| exp ? exp : exp
| ! exp
| exp == exp
| exp != exp
| exp < exp
| exp > exp
| exp >= exp
| exp <= exp
| exp && exp
| exp || exp
| | exp |
| exp + exp
| exp − exp
| exp * exp
| exp / exp
| exp % exp
| exp >> exp
| exp << exp
| exp . id
| exp [exp]
| exp [exp .. exp]
| exp \+ (id_list)
| exp \− (id_list)
| (type) exp
| md5 (exp)
| **getcookie** (exp)
| **setcookie** (exp , exp)
| **getenv** (exp)
| **getYear** (exp)

| **getMonth** (exp)
| **getDay** (exp)
| **getHour** (exp)
| **getMinute** (exp)
| **getSecond** (exp)
| **getWeekday** (exp)
| **open** (exp , stringconst)
| **open** (exp , exp)
| **print** (exp , exp)
| **println** (exp , exp)
| **scan** (exp , id)
| **scanln** (exp)
| **eof** (exp)
| **close** (exp)
| **fileerror** (exp)
| **random** (exp)
| **system** (exp)
| exp < [plug_list]
| **rawhtml** (exp)
| **track** (exp)
| **match** (exp , exp) [getinput_list]
| **get** (exp)
| **get** (exp) [cgiinput_list]
| **post** (exp)
| **post** (exp) [cgiinput_list]
| **sort** (exp)
| **sort** (exp ; id_list)
| **cart** (exp , exp)
| **factor** (exp_list) stm
| **factor** (exp_list ; id_list) stm
| (compound_stm)

tupleexp_list : tupleexp⊛
tupleexp : id = exp
plug_list : plug⊕
plug : id = exp
cgiinput_list : cgiinput⊕
cgiinput : exp = exp

---

boolconst : BOOLCONST

intconst : INTCONST
floatconst : FLOATCONST
charconst : CHARCONST
stringconst : STRINGCONST
id_list : id⊕
id : IDENTIFIER
| ` id
| id ~ id

Legend

NT⊛ : Zero-or-more repetitions of the nonterminal NT.
NT⊛ : Zero-or-more **comma separated** repetitions of the nonterminal NT.
NT+ : One-or-more repetitions of the nonterminal NT.
NT⊕ : One-or-more **comma separated** repetitions of the nonterminal NT.

<Operators>

Expressions

The precedences are listed in *decreasing* precedence order. For instance, multiplication binds stronger than addition, which means that "`1+2*3`" is understood as "`1+(2*3)`"

Operator	Description	Associativity
`(__)`	parentheses	N/A
`(type) __`	type conversion	N/A
`|__|`	vector size	N/A
`|__|`	relation size	N/A
`|__|`	string size	N/A
`({__})`	expression-statement	N/A
`__[__]`	string indexing (*from 0 to |s|-1*)	N/A
`__[__]`	vector indexing (*from 0 to |v|-1*)	N/A
`__[__..__]`	substring	N/A
`__[__..__]`	subvector	N/A
`__.__`	tuple member reference	left
`!__`	logical NOT	N/A
`-__`	unary minus	N/A
`+__`	unary plus	N/A
`++__`	pre increment	N/A
`--__`	pre decrement	N/A
`__++`	post increment	N/A
`__--`	post decrement	N/A
`__<[__]`	document plug	N/A
`__*__`	multiplication	left
`__/__`	integer division	left
`__/__`	float division	left
`__%__`	modulo	left
`__\+(__)`	tuple project	N/A
`__\-(__)`	tuple project complement	N/A
`__+__`	addition	left
`__+__`	string concatenation	left
`__+__`	vector concatenation	left
`__-__`	substraction	left
`__<<__`	left tuple overwrite	left
`__>>__`	right tuple overwrite	left

`___<___`	less than	N/A
`___>___`	greater than	N/A
`___<=___`	less than or equal	N/A
`___>=___`	greater than or equal	N/A
`___==___`	equal	left
`___!=___`	not equal	left
`___&&___`	lazy logical AND	left
`___||___`	lazy logical OR	left
`___?___:___`	conditional operator	right
`___=___`	assignment	right
`___+=___`	addition, then assignment	right
`___-=___`	substraction, then assignment	right
`___*=___`	multiplication, then assignment	right
`___/=___`	division, then assignment	right
`___%=___`	modulo, then assignment	right
`___=<[___]`	plug, then assign	right

lazy

means that if the result of an expression can be determined by evaluating the left operand only, then the right operand is not evaluated. This makes a difference when the right operand has side-effects or does not terminate.

left

means that the given operator is **left-associative**. For instance, "`x+y+z`" will be parsed as "`(x+y)+z`".

right

means that the given operator is **right-associative**. For instance, "`x=y=z`" will be parsed as "`x=(y=z)`".

<Scope>

Scope

The scope rules in <bigwig> are static (or lexical) for both functions and variables. The following constructs cause a new scope frame:

Scope frame	#passes
service	2
session	2
schema	2
function	2
function prototype	2
compound statement	1
factor	1
all	1
is	1

<Types>

[Basic Types | Composite Types | Type Modifiers | Initial Values | Conversion | Coercion]

[void | bool | int | float | char | string | time | file | html | tuple | relation | vector]

Basic Types

void

The type **void** is special and purely used in association with the declaration of functions not returning anything (a.k.a. procedures). It is not possible to create a variable of type **void**.

bool

There are only two possible values of type **bool** (a.k.a. `boolean`), namely "**true**" and "**false**". The type would thus ideally occupy one bit at runtime. However, for reasons of efficiency it is represented internally by an **int**.

int

The integer type, **int**, is for signed numbers. It corresponds directly to **C**'s **int**'s, thus its precision is implementation specific. The size will typically reflect the natural word size of the host machine, the size is typically 32 (or 64) bits, although it is guaranteed to be at least 16 bits.

float

The **float** type is for signed pseudo real numbers. They correspond to **C**'s `double` the precision of which is implementation specific. They will typically be 64 (or 128) bits.

char

The values of type **char** are (ascii) characters that occupy 8 bits at runtime.

string

A **string** is a basic type, the values of which are arbitrary sequences of **char**s starting from zero. All **string**s are at runtime represented as indices into a local *string-pool* holding the actual **`string`** along with its length. Thus operations such as string-compare "`s==t`" and string-length "`|s|`" takes constant (not linear!) time. The lexicographic order operators "<" and ">" take linear time.

time

The values of type **time** are legal points in time. That is, a (gregorian calender) date and an hour-minute-second time of day. Illegal times, such as "`1999/02/29, 20:61:30`" are automatically converted to legal ones ("`1999/03/01, 21:01:30`"). The predefined function **now()** returns the current time. All **time** values must be within the following range [`1970/01/01, 00:00:00..2038/01/19, 03:14:07`]. Any attempts to construct a **time** not in this interval will result in the special **time** value called **notime** which is also the default value for the type **time**. See Time for more information.

file

The values of type **file** are *file-handles* with a limited number of operations. See Files for more information.

html

html is a special type in <bigwig> . A value of type html is an html document with a number of named gaps that can be plugged with strings or other html documents at runtime. The html documents are represented in a very compressed format with maximum sharing and where the plug operation takes constant time. See Dynamic Documents for more information.

Composite Types

tuple

The **tuple** type is defined relative to a schema which is essentially a mapping from identifiers to basic types. A tuple value is thus a mapping from identifiers to basic values corresponding to the names and types of the associated schema. The tuple identifiers are commonly referred to as attributes. See Database for more information.

relation

A relation is a set of tuples with no notion of order and where no value is present twice (any redundant values are implicitly removed). The **factor** operation is available for traversing relations. See Database for more information.

vector

There are two kinds of Vectors (a.k.a. lists or arrays) in <bigwig>; basic vectors and tuple vectors. Basic vectors are capable of storing lists of basic values, while tuple vectors contain lists of tuples. Both kinds can be multi-dimensional. See Database for more information.

Type Modifiers

const

Any type can be prefixed with the type modifier **const**, that will generate a type that cannot be assigned values.

shared

This modifier can be applied to a declarations to make them persistent and *shared* among all sessions in the service.

Initial Values

<bigwig> initializes all variables upon declaration, hence all types have associated initial values. The precisions are inherited by the compilation target language (C). These values can be seen in the table below:

Type	Min. prec.	Typ. prec.	Initial Value	Order	Equality
void	N/A	N/A	N/A	N/A	N/A

bool	1 bit	1 bit	**false**	N/A	yes
int	16 bits	32 bits	0	numeric	yes
float	32 bits	64 bits	0.0	numeric	yes
char	8 bits	8 bits	' ' (space)	ascii	yes
string	N/A	N/A	""	lexicographic	yes
time	32 bits	32 bits	**notime**	numeric	yes
file	N/A	N/A	N/A	N/A	no
html	N/A	N/A	**<html></html>**	N/A	no
tuple	N/A	N/A	**tuple** { ... * }	N/A	yes
relation	N/A	N/A	**relation** { }	N/A	no
vector	N/A	N/A	**vector** { }	N/A	no

*) The individual attributes in the tuple will have initial values dictated by their types. The types of attributes in a tuple are all required to be basic.

Type Conversion

<bigwig> provides a simple means for explicit conversion. The legal conversions and their semantics are listed below. All types (except **file**) can be converted to and from **string**. Conversion is specified by prefixing a given expression with the name of the type in parentheses.

- **bool** can be converted to:
  - **string**: true and false become "true" and "false", respectively.
  - **html**: true and false become <**html**>true</**html**> and <**html**>false</**html**>, respectively.
- **int** can be converted to:
  - **float**: Straightforward. For instance, 42 becomes 42.0.
  - **char**: Integers are converted to chars according to their ascii value (modulo 256). For instance, 65 becomes 'A'.
  - **string**: Straightforward. For instance, 42 becomes "42".
  - **html**: Straightforward. For instance, 42 becomes <**html**>42</**html**>.
- **float** can be converted to:
  - **int**: Floating point numbers are truncated to integers. For instance, 1.75 becomes 1.
  - **string**: Straightforward. For instance, 3.14 becomes "3.14".
  - **html**: Straightforward. For instance, 3.14 becomes <**html**>3.14</**html**>.
- **char** can be converted to:
  - **int**: Chars are converted to ints according to their ascii value. For instance, 'A' becomes 65.
  - **string**: All chars are converted to a strings of length one. For instance, 'x' becomes "x".
  - **html**: Straightforward. For instance, 'x' becomes <**html**>x</**html**>.

- **`string`** can be converted to:
  - **`bool`**: `"false"` becomes `false`, the rest becomes `true`.
  - **`int`**: Strings only containing ciphers (and white-spaces) are translated directly into integers. All other strings become `0`. For instance, `" 42"` becomes `42`
  - **`float`**: As above, except that the string may contain one dot. For instance `"3 . 14"` becomes `3.14`.
  - **`char`**: Empty strings become `'\0'`, the rest become the first character in the string. For instance, `"hello"` becomes `'h'`.
  - **`time`**: Currently Strings must be `"yyyy/mm/dd, hh:mm:ss"`.
  - **`html`**: Strings are translated into html by escaping angled brackets. For instance, the string `"<bigwig>"` is turned into `"<html><bigwig></html>"`. Alternatively, if the brackets should not be escaped, the type conversion construct **rawhtml** may be used. Use with caution!
  - **`tuple`**: The string will be unserialized. **Note:** *Not yet implemented.*
  - **`relation`**: The string will be unserialized. **Note:** *Not yet implemented.*
  - **`vector`**: The string will be unserialized. **Note:** *Not yet implemented.*
- **`time`** can be converted to:
  - **`string`**: Returns `"yyyy/mm/dd, hh:mm:ss"`.
- **`tuple`** can be converted to:
  - **`string`**: The tuple will be serialized.
- **`relation`** can be converted to:
  - **`string`**: The relation will be serialized.
  - **`vector`**: Straightforward. The order of the individual tuples is unspecified.
- **`vector`** can be converted to:
  - **`string`**: The vector will be serialized.
  - **`relation`**: Straightforward. Identical entries are removed.

Coercion

`<bigwig>` generally does not coerce expressions. The exceptions to this rule are plug, receive, and string concatenation. Any basic value can be plugged into a document gap, causing a coercion from this type to the html type as explained under conversion above. Similarly, at `receive` the assigned variable can have any basic type (wherever a basic type is expected, see Form Input Table).

<Core Language>

[Parameter Mechanisms | Garbage Collection | Prototype Functions | Miscellaneous]

Parameter Mechanisms

All assignments (including implicit assignments of actual parameters to formal parameters induced by function calls) involving basic types in <bigwig> are *call-by-value*. Those involving composite types are performed by *copy-on-write*, which is semantically equivalent to *call-by-value*, only much more efficient.

Garbage Collection

All values in <bigwig> are garbage collected when they are no longer *live* (reachable by program variables). This happens incrementally, automatically, and transparently via a built-in *reference counting garbage collector*.

Prototype Functions

<bigwig> is equipped with the possibility of interacting with functions written (externally) in **C**. This is done through *prototype functions* (i.e. function declarations with no body). Whenever the <bigwig> compiler encounters such a declaration, it will assume that the body of the function is provided by some **C** code. This can sometimes be useful, when constructions that are not directly provided by <bigwig> are needed. [See the getting started tutorial].

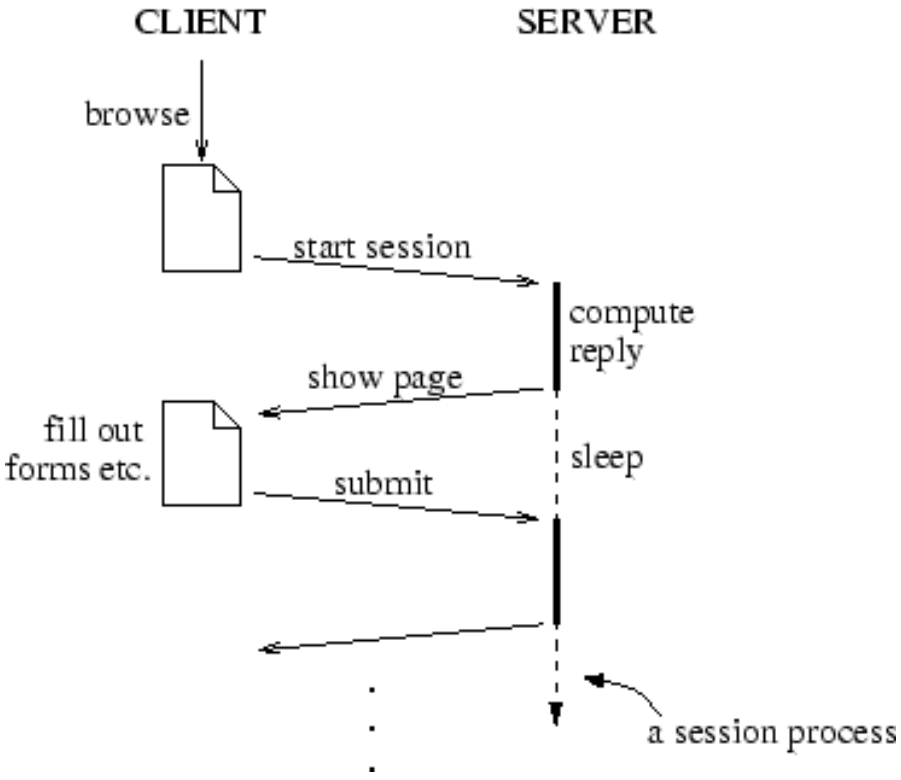
Miscellaneous

service

A <bigwig> program is also referred to as a Web service and begins with the keyword **service**. A service is comprised of a number of toplevel declarations plus a number of sessions.

session

Sessions are the entry points to the Web service, much like the main routine in C and Java programs. However, unlike C and Java programs, a <bigwig> service may have more than one **session**. A **session** contains a number of toplevel declarations plus a sequence of statements that is executed when the **session** is *invoked* (via a CGI request). This sequential action may at various points chose to interact with the client who invoked the **session** (almost reversing the roles of client and server), asking for values to be entered and submitted. Unlike CGI programs, <bigwig> sessions preserve the state across interactions. A **session** gives rise to a *session process* on the Web server as depicted below:



---

getenv

This built-in function takes a **string** argument naming an environment variable and returns the (**string**) value of this variable.

random

This built-in function takes an int value and returns a (pseudo-) random int value between (including both end points) zero and the value given minus one.

system

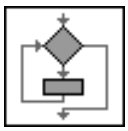
This built-in function is similar to C's system function. It takes a string which is executed in a shell and returns the (integer) exit status produced by the shell. **Caution:** be careful when using this command!!!

Expression-Statements

Statements can now be embedded in expressions (*exp* ::= '(' *compound_stm* ')') as in Gnu C. The result is the value of the last statement if it is a statement-expression (*stm* ::= *exp* ';') and **void** otherwise.

typeof

This construct takes an expression in parentheses. The type of the construct is equal to the type of the expression. Note that the expression is solely used for determining the type and is thus never run. The construction is mostly intended for use via the syntax macros.



<Control Structures>

[if-else | while | for | return | exit | switch | wait]

---

if

stm ::= **if** (*exp*) *stm*
stm ::= **if** (*exp*) *stm* **else** *stm*

The **if** / **if-else** statement provides the simplest form of conditional execution. The expression (usually called the *condition*) that must be of type **bool** is evaluated. If it evaluates to **true**, the statement immediately following the expression in parentheses is executed. It is usually called the *then-statement*. If the expression evaluates to **false**, the statement following the **else** (if any) is executed. This statement is commonly referred to as the *else-statement*. An **else** will always be bound to the closest else-less previous **if** statement.

while

stm ::= **while** (*exp*) *stm*

The simplest form of iteration (or looping) is provided by the **while** statement. The expression (called the *condition*) that must be of type **bool** is evaluated. If it evaluates to **false**, the **while** statement has been executed. If on the other hand it evaluates to **true**, the while statement (usually called the *body of the while*), is executed. Once executed, control is passed once again to the entire **while** statement, and the procedure is repeated.

for

stm ::= **for** (*exp*? ; *exp*? ; *exp*?) *stm*

The **for** statement is a variant of **while**. The three expressions (called the *initialization*, *condition*, and *increment*) are all optional. If present, the *condition* must be of type **bool**. Initially, the *initialization* expression is evaluated for its side-effects. Hereafter, as with **while**, the *condition* is evaluated and if **false**, the **for** statement has been executed. If the *condition* evaluates to **true**, the statement (called the *body of the for*) is executed. Once the statement has been executed, the *increment* expression is evaluated for its side-effects and control is passed again to the entire **for** statement and the procedure is repeated, ignoring the *initialization* expression. The statements "**for** (E_1; E_2; E_3) S" and "{ E_1; **while** (E_2) { S; E_3; } }" are completely equivalent.

return

stm ::= **return** ;
stm ::= **return** *exp* ;

The **return** statement serves two purposes in <bigwig>. The return without an expression is used for returning from procedures (functions with return type **void**). The **return** statement has the effect that control is passed to the point immediately following the point that called the procedure (function). The second is used for returning values from functions. The type of the expression, which is required to be the same as the return-type of the function, is evaluated and

returned to the point where the function was called. Hereafter control resumes at this point. Another use of the **return** statement is in conjunction with the **factor** statement.

exit

stm ::= **exit ;**
stm ::= **exit** *exp* **;**

The **exit** statement causes the session to terminate. If an expression is supplied, it is required to be either of type **html** or **string**. If the type is **html**, the document resulting from the evaluation of the expression will be shown to the client just prior to termination. If the type is **string**, the expression is assumed to evaluate to an URL. The page shown will contain a jump to the specified URL. If on the other hand, no expression is supplied, a default termination message will be shown.

switch

stm ::= **switch** (*exp*) { *switchbranch-list* }
switchbranch-list ::= *switchbranch*$^+$
switchbranch ::= **case** *exp* **:** *stm-list* **break;**
switchbranch ::= **default** **:** *stm-list* **break;**

The **switch** statement is basically a multi-dimensional **if** statement. The expression in parentheses (called the *switch-expression*) is evaluated to a value. Then, the *case-expressions* are evaluated, in order, one by one, until the value of one of them is equal to the *switch-expression*. At this point the rest of the *case-expressions* are ignored and control is past to the corresponding **case**'s statement after which the **switch** statement has completed. All the involved expressions must have the same basic type. The **switch** statement works with all basic types, except **html** and the expressions involved do not have to be constant (as in, for instance **C** or **Java**). A **switch** statement is allowed to have maximum one **default** *branch* and if such a *branch* exists, it must be specified after all the case branches. If none of the expressions matched the *switch-expression* and there is a default branch, the *default-statement* is executed. The **break** keyword is required at the end of each *branch*.

wait

stm ::= **wait** *id* **;**
stm ::= **wait** { *waitbranch-list* }
waitbranch-list ::= *waitbranch*$^+$
waitbranch ::= **case** *id* **:** *stm-list* **break;**
waitbranch ::= **timeout** *exp* **:** *stm-list* **break;**

The **wait** statement is special to <bigwig>. It provides the interaction with the *runtime controller*. There are two kinds of **wait** statements, namely the *short wait* corresponding to the first two productions above, and the *general wait* corresponding to the third production above.

The short wait:
When executing a short wait statement "**wait** L;" (where L is some label defined as a constraint), the session thread will contact the runtime controller, requesting permission to pass the label L. If the entire service is in a state where it is *safe* for the session thread to pass the label L (i.e. without violating the safety contraints, meaning that L is *enabled*), the runtime controller will grant the session thread permission to proceed. Otherwise the session is

suspended (even indefinitely), until it is safe for it to proceed.

The general wait:
The general wait is like the short wait, except that it allows the session thread to branch on the state of the runtime controller. The semantics is that permission to proceed is granted when *one* of the labels is passed (enabled). Execution proceeds afterwards at the **case** corresponding to the label passed. A general wait can also contain one **timeout** branch (where the expression evaluates to an integer). The expression is evaluated to a time bound (in seconds) and if the runtime controller cannot grant permission to continue within this bound, execution resumes with the timeout statement-list. Only one **timeout** is allowed in a **wait** statement and it must be specified after all the **case**s.

<Files>

[open | close | fileerror | print | println | scan | scanln | eof]

The values of the file type are *file handles* with only a limited number of operations.

Opening and closing files

exp **::=** **open** (*exp*)

exp **::=** **open** (*exp* , *exp*)

The first argument is a string designating a file on the server's file system. The second argument specifies the open mode and must be on of the strings "read", "write", or "append", defaulting to "read" if not present.

exp **::=** **close** (*exp*)

Will close the file associated with the file handle given as argument. The expression will return void.

exp **::=** **fileerror** (*exp*)

Takes an expression of type `file` and returns a boolean that is `true` if a previous open on the `file` was successful and `false` if not.

Printing in files

exp **::=** **print** (*exp* , *exp*)

The string specified in the second argument is printed to the file designated by the first argument. The file must be in either write or append mode. The expression will return void.

exp **::=** **println** (*exp* , *exp*)

The string specified in the second argument plus a newline character is printed to the file designated by the first argument. The file must be in either write or append mode. The expression will return void.

Scanning in files

exp **::=** **scan** (*exp* , *id*)

Scan will take two arguments, a file handle and a format. It will 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. The file must be in read mode. The expression will return what was scanned from the file.

exp **::=** **scanln** (*exp*)

Scans and returns one line from the file specified in the argument. After this, the file pointer is advanced accordingly. The file must be in read mode.

exp **::=** **eof** (*exp*)

Returns a **true** if the file designated by the argument is at the end-of-file mark, **false** if not. The file must be in read mode.

<Macros>

[Macro Syntax | Packages | Overloading | Alpha Conversion | Order of Expansion | Nonterminals | Macro Libraries]

Macro Syntax

package : req_ext* macro*

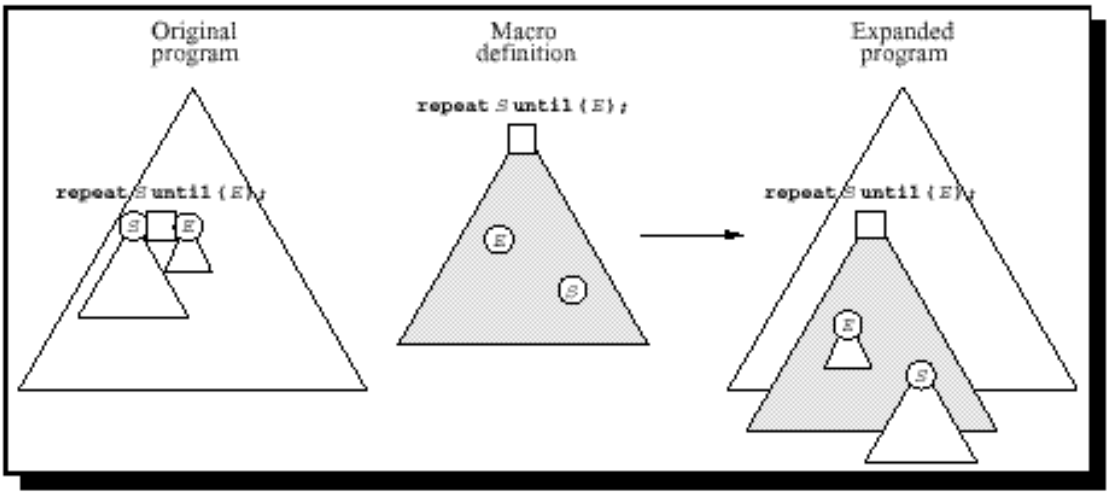
req_ext : **require** <URL> | **require** stringconst
| **extend** <URL> | **extend** stringconst

macro : **syntax** <*nonterm*> id param* **::=** { body }
| **metamorph** <*nonterm*> id **-->** param* **::=** { body }

param : token
| <*nonterm* id>
| <id: *nonterm* id>

A syntax macro has four constituents: a nonterminal *result type*, an identifier *naming* the macro, a *parameter list* specifying the invocation syntax, and a *body* that must comply with the result type.

A metamorphism has the four constituents: a nonterminal *result type*, an identifier *naming* a user defined nonterminal (left-hand side meta grammar production), a *parameter list* specifying the invocation syntax, and a *body* that must comply with the result type.



Syntax macros: operators on parse trees.

Packages

A *package* is a *file* containing macro definitions. A package is viewed as a *set*, that is, we use *two*

pass scope rules where all definitions are visible to each other and the order is insignificant. A dependency analysis intercepts and *rejects* cyclic definitions.

A package may **require** or **extend** other packages. Consider a package *P* that contains a set of macro definitions *M*, requires a package *R*, and extends another package *E*. The definitions visible inside the bodies of macros in *M* are $M \cup R \cup E$ and those that are exported from *P* are $M \cup E$. Thus, **require** is used for obtaining *local* macros.

There is no scoping mechanism for undeclaring a macro.

Overloading

Macros may be overloaded meaning that two macro definitions the same identifier name but have different invocation syntax (different parameters) and different bodies. However, macros with the same name must have same nonterminal return type.

When there are more than one macro with the same name, we base the invocation selection on the concept of *specificity* which is independent of the macro definition order. This is done by gradually challenging each parameter list with the input tokens. There are three cases for a challenge:

- if a list is empty, then it always survives;
- if a list starts with a token, then it survives if it equals the input token; and
- if a list starts with an argument <*N id*>, then it survives if the input token belongs to first(*N*) in the host grammar.
- if a list starts with an argument <*M*: *N a*>, then it survives if the input token belongs to first(*M*) in the metamorph grammar.

Several parameter lists may survive the challenge. Among those, we only keep the most *specific* ones. The empty list is always eliminated unless all lists are empty. Among a set of non-empty lists, the survivors are those whose first parameter is maximal in the ordering $p \sqsubset q$ defined as $\Phi(q) \subset \Phi(p)$, where Φ is:

		Φ: *param* → 2^{TOKEN}	
Φ(*token*)	=	{*token*}	
Φ(<*N id*>)	=	first(*N*)	...in the host grammar
Φ(<*M*: *N a*>)	=	first(*M*)	...in the metamorph grammar.

The tails of the surviving lists are then challenged with the next input token, and so on.

Note that the strategy is *greedy*, since it prefers to continue with longer parameter lists.

Alpha Conversion

The body of a macro constitutes a closed scope. Free identifiers in a macro body are alpha converted to avoid identifier clashes and shading after the expansion. Alpha conversion can also be supressed using the identifier prefix operator: ` (backping), which only makes sense in the bodies of macro definitions.

The predicate α determines if an identifier will be α-converted:

- α(`i) = false;
- α(i~j) = α(i) $\wedge$ α(j);
- α(<i>) = false, if <i> is a macro argument of type *id*; and
- α(i) = true, otherwise.

Order of Expansion

In the presence of nested macro invocations, we expand the innermost first yielding *applicative order of reduction*. This results in a *call-by-value* expansion semantics.

Nonterminals

The nonterm above can be any of the 55 syntactic categories below:

[argument | argument_list | attr | attribute | attribute_list | boolconst | charconst | compound_stm | constraint | constraintbody | constraintbody_list | decl | decl_list | defattr | exp | exp_list | field | field_list | floatconst | format | formula | getinput | getinput_list | html | htmlbody | htmlbody_list | identifier | identifier_list | intconst | label | mode | mode_list | modifier | modifier_list | plug | plug_list | regexp | regexp_list | schema | service | session | stm | stm_list | stringconst | switchbranch | switchbranch_list | toplevel | toplevel_list | trigexp | trigger | tupleexp | tupleexp_list | type | waitbranch | waitbranch_list]

Macro Libraries

"std.wigmac" (*the standard macro library*).

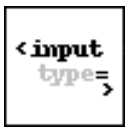
It contains constant definitions, control structures, and high-level concurrency abstractions.

"sql.wigmac" (*the SQL macro library*).

It contains SQL statements for operating on relations.

"enum.wigmac".

It contains the "enum" (metamorph) construction.



<Form Input>

[button | checkbox | file | hidden | image | password | radio | reset | submit | text | <textarea> | <select> | <tuple> | <continue>]

Form input table

tag	type	name	value	submits	multiplicity	return type
<input>	button	●	●	no	no	basic
<input>	checkbox	●	●	no	yes	list { value }
<input>	file	●	-	no	no	basic/tuple*
<input>	hidden	●	●	no	no	basic
<input>	image	●	-	yes	no	tuple { x,y }
<input>	password	●	○	no	no	basic
<input>	radio	●	●	no	yes	basic
<input>	reset	-	○	no	-	-
<input>	submit	C	○	yes	yes	basic
<input>	text	●	○	no	no	basic
<textarea>		●	○	no	no	basic
<select>		●	-	no	no	basic
<select multiple>		●	-	no	no	list { value }
<tuple>**		●	-	no	yes	list { ... }
<continue>**		C	○	yes	yes	basic

● Attribute *required*.

○ Attribute *optional*.

C The name is implicitly "continue".

*) The file input field may return a tuple with three strings named `filename`, `contents`, and `type`.

**) These tags are special to <bigwig>. They are *not* part of standard HTML. They are compiled to other tags and JavaScript; The client never sees these tags.

multiplicity

Multiple input elements allowed with the same name (value of the `name` attribute).

basic

Can be either: **bool**, **int**, **float**, **char**, **string**, and **time**. An implicit coercion from type string will take place.

list

Can be either a **vector** or a **relation**, with schemas containing the mentioned attribute names

(all of which are of basic type).

`button`

Straightforward.

`checkbox`

Multiple checkboxes with the same name is allowed, they will together return a list with the values of all the boxes checked.

`file`

May be received as either a **string** or a **tuple** with three strings named `filename`, `contents`, and `type`. In the former case, the value received will be the name of the file the client selected. In the latter case, the tuple three fields will respectively receive the name of the file, the contents of the file, and the mime type of the file.

`hidden`

Straightforward.

`image`

Will return the x and y coordinates of the point clicked in the image as a **tuple** with two attributes "x" and "y". The click will cause the page to be submitted.

`password`

The `text` field always returns a *basic* type and is required to have a `name` attribute. If a `value` attribute is supplied, this will be the text initially present in the field. A multitude of sophisticated (optional) attributes are available for manipulating the field's functionality and look-and-feel (see PowerForms).

`radio`

Multiple radio buttons with the same name is allowed, but as in HTML only one of them will be returned.

`reset`

Will cause the form to reset.

`submit`

Will submit the form.

`text`

`text` is the default type for form input elements. The `text` field always returns a *basic* type and is required to have a `name` attribute. If a `value` attribute is supplied, this will be the text initially present in the field. A multitude of sophisticated (optional) attributes are available for manipulating the field's functionality and look-and-feel (see PowerForms).

`<textarea>`

Straightforward.

`<select>`

If the "multiple" tag is present, the field will return a list of the options selected. If not, it will return the option selected as a basic value.

`<tuple>`

The tuple input field allows information of dynamically varying size to be received and

collated into **vector**s or a **relation**s on the server-side. Tuples baring the same name are viewed as a single input field. This extension allows multiple occurrences of the same input field name to appear in a document. When the field values are received, they are automatically transformed into tuples in a **vector** or a **relation** with corresponding schema. Since <bigwig> does not have support for higher-order relations, tuples cannot be nested.

<continue>

The continue field is added as a convenient way of submitting values. A continue field cannot have a name, instead the keyword **continue** is available for value reception in (show-)**receive** statements. The value attribute is optional, but must be received if present. The **continue** field has two variants "type=button" and "type=text" (default), controlling the appearance of the submission field. That is, whether it should be a button or a text link. The continue field is compiled into a button or an HTML anchor with embedded JavaScript that will take care of submitting the form when clicked.

Note: This will only work for JavaScript enabled browsers.

<Formats>

[Syntax | Semantics | Format Uses | PowerForms | Ignoreformats | String Matching | File Scanning]

Syntax for format related constructs

format	::= **format** *id* **=** *regexp* **;**	format definition
regexp_list	::= regexp$^{\oplus}$	
regexp	::= *id*	format reference
	| *stringconst*	constant
	| **anychar**	anychar
	| complement **(** *regexp* **)**	complement
	| **concat (** *regexp_list* **)**	concatenation
	| **fix (** *intconst* **,** *intconst* **)**	fixed integer interval
	| **intersection (** *regexp_list* **)**	intersection
	| **range (** *charconst* **,** *charconst* **)**	character range
	| **regexp (** *stringconst* **)**	(perl) regexp
	| **relax (** *intconst* **,** *intconst* **)**	relaxed integer interval
	| **star (** *regexp* **)**	kleene's star
	| **union (** *regexp_list* **)**	union
	| [*id* = *regexp*]	record regexp

Semantics for format related constructs

id (*format reference*)

This rule is for referencing regular expressions defined by other formats. The effect of referring to another regular expression is as if the expression had been written on the spot directly. Due to the non-recursive nature of regular expressions, these id's may not be used recursively. The same thing goes for mutual recursion.

stringconst (*constant*)

This regular expression will only match the string *constant* itself.

anychar

This is a constant regular expression that will match any `ascii` character.

complement

This regular expression will match anything the regular expression argument to this construct does not. It constitutes complement with respect to `ascii*`.

concat

This will be the regular expression corresponding to the concatenation of the regular expression arguments.

fix

This will be the regular expression matching any number in the interval (both end-points included) specified by the two intconst arguments.

intersection

This will be the regular expression corresponding to the intersection of the regular expression arguments. That is, a string is matched if and only if it is matched by all of the regular expression arguments.

range

This regular expression will only match characters in the interval (both end-points included) specified by the two charconst arguments.

regexp

This regular expression allows regular expressions to be specified in Perl style. Characters escaping requires prefixing of two backslash characters.

relax

This will be the regular expression matching any number in the interval (both end-points included) specified by the two intconst arguments. It will however *not* match numbers prefixed with zeros (as in `"007"`).

star

This will be the regular expression corresponding to kleene's star of the supplied regular expression. That is, any number (including zero) of repetitions of the regular expression supplied.

union

This will be the regular expression corresponding to the union of the regular expression arguments. That is, a string is matched if and only if it is matched by at least one of the regular expression arguments.

record

This regular expression is the identity on the regexp argument specified to the right of the equality character, but with the exception of one side-effect. The effect is that when used for testing strings the regexp is ``recorded'' and becomes available for reception using the name specified in the identifier.

Format Uses

The **format** construct is available for defining regular expressions that can be used for three things in `<bigwig>`:

- "PowerForms" - restricting form inputs in html documents,
- String matching (against regular expressions), and
- File scanning.

See also the PowerForms **Stand-alone** package homepage

"PowerForms" - Restricting form input in html documents

The primary usage of **format**s is in conjunction with "text" or "password" input fields in html forms. The syntax for adding a format to a text input field is:

<input type=text format=*id* **name=***defattr* **... >**

Such "enhanced" text fields (a.k.a. *power fields*) will be compiled into standard html text-fields with JavaScript ensuring increased functionality. The effect of adding a format to a text field is that the client will check (through JavaScript interpreting deterministic finite automata) that the contents of the text field complies with the associated regular expression. As long as there are text fields that do not comply with their associated formats, the html page cannot be submitted and a suitable message will be given. This form input validation will be performed incrementally (that is, as the client enters the data) and the results will be visualized in a dynamically updated image next to the input field. At any point in time, this image will show one of three images (corresponding to the current state of the deterministic finite automaton):

"../formats/green.gif":

when the input fields current string is in the language defined by the regular expression.

"../formats/yellow.gif":

when the input fields current string is in the prefix closure of the language defined by the regular expression, but not in the language itself.

"../formats/red.gif":

when the input fields current string is neither in the prefix closure of the language defined by the regular expression, nor in the language itself.

The three images default to a traffic-light style as shown below:

Example:

The field accepts strings with zero, three, or six characters.

For reasons of security the fields are double checked at reception time on the server-side. The following options are available:

attribute	effect
green	The value is the text shown in the browser's status bar when the input is in the language defined by the regular expression.
red	The value is the text shown in the browser's status bar when the input is neither in the language defined by the regular expression, nor in the prefix of this language.
yellow	The value is the text shown in the browser's status bar when the input is in the prefix of the language defined by the regular expression, but not (yet) in this language itself.

---

ignoreformats - Overriding Form Input Validation

The submission blockage caused by fields not complying with their associated formats can be overridden using the **ignoreformats** attribute. This attribute will only make sense for buttons

that cause a page to be submitted (i.e. submit-buttons, <continue>, and input-images).

String Matching

For all defined formats, the **match** construct is available for testing whether a string complies with the associated regular expression defined by the format. The result will be a boolean stating whether or not the string is in the language induced by the regular expression. Any recordings are available as the right hand sides of assignments in the comma-separated list enclosed in square brackets.

File Scanning

Formats can also be used for scanning in files. Syntactically, **scan** takes an expression designating a file and an identifier denominating a format.

exp	::= **scan (*exp* , *id*)**		file scan

A call to **scan** will return the longest (possibly empty) string in the regular expression defined by the format at the current position in the file specified. After this, the current file position will be updated accordingly.

<Dynamic Documents>

[Introduction | Document Values | Static safety | Syntax | Semantics | Flow Join Requirements | HTML Prototypes | Code Gaps]

Introduction

In addition to the usual general-purpose programming-language types, (**bool**, **int**, **float**, and so forth), <bigwig> is equiped with a more domain specific type, namely that of **html**. The type **html**, ranges over html documents that may contain named gaps that act as placeholders for either HTML fragments or attributes in tags. HTML documents are first-class values that may be computed and stored in variables. The documents are represented in a very compressed format, with maximum sharing and where the plug operations takes constant time only. A flow-sensitive type checker ensures that documents are used in a consistent manner.

Document Values

An html document value is comprised of three distinct constituents, *gaps*, *fields*, and *text*:

- **Gaps**
  Gaps are named placeholders that can be plugged with other html documents or strings at runtime using the plug operator. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. There are two kinds of gaps, namely *document gaps* (html gaps) and *attribute gaps* (string gaps).
  - **Document gaps:** A document gap is written

    `<[gap_name]>` or `<["gap_name"]>`

    and acts as a named placeholder for other html documents or strings.
  - **Attribute gaps:** An attribute gap is written

    `<tag ... [gap_name] ...>` or `<tag ... ["gap_name"] ...>`.

    The important difference from the above is that these gaps are written inside tags, meaning that there is at least one attribute between the open angled bracket '<' and the actual gap. Such attribute gaps act as placeholders for strings. They cannot be plugged with html documents.

  Gap requirements: No html documents may have the same named gap present more than once.
- **Fields**
  Fields or rather "input fields" are document constituents that allow for information to be somehow entered or selected, typically using a browser. Thus, when documents containing input fields are shown (via the **show** construct) they will have an information flow back to the service that must be received using the **receive** construct.

Field requirements: The fields must meet the requirements mentioned in the form input elements table.

- **Text**
  This is basically "the rest" of the document, which is everything not containing gaps nor input-fields. This is the actual verbatim text in the document with all tokens verbatim (including whitespaces) along with the formatting.

The default value for variables of type **html** is the empty html document **<html></html>**, containing no gaps, no fields, and no text.

Static safety

Using a specialized data-flow analysis, we check programs and provide static safety. Once a program has been checked and found to be well-typed, we can provide static safety, that is, we can statically guarantee that:

- all documents constructed meet the gap and field requirements above;
- the document to be plugged has the gap specified in the plug operation;
- the number receiving program variables in a show-receive call equals the number of document input fields; and
- the types of receiving program variables in a show-receive call match the corresponding input field kinds.

Static safety is provided by introducing a somewhat complicated notion of document type. However, these types are not explicitly written by the programmer, but are inferred by the compiler using a global flow analysis. A document type has two distinct components: a *gap map* and a *field map*. The gap map describes which gaps are present in a document along with their kinds (html, string) and, similarly, the field map describes its input fields and their kinds (text, radio, checkbox, etc...).

Our interprocedural monovariant first-order forward flow analysis associates sound gap and field information for each document variable to each program point. Based on this information we can then determine if a program is type correct. If it is we can provide static safety, that is, we can statically guarantee that the four properties listed above will in fact hold.

Syntax for html related constructs

exp ::=	**<html>** *htmlbody_list* **</html>**	constant
	id	variable
	exp **=** *exp*	assignment
	exp **<[** { *id* **=** *exp* }$^{\oplus}$ **]**	plug
	id **(** { *exp* }$^{\circledast}$ **)**	function call
	track (*exp* **)**	track
stm ::=	**flash** *exp* **;**	flash
	show *exp* **;**	show

	show *exp* **receive** **[** { *exp* **=** *id* }⊛ **]** **;**	show-receive
	show *exp* **timeout** *stm*	show/timeout
	show *exp* **receive** **[** { *exp* **=** *id* }⊛ **]** **timeout** *stm*	show/timeout
	exit *exp* **;**	exit

Legend:

{ *X* }⊛: Zero-or-more comma-seperated occurences of *X*.

{ *X* }⊕: One-or-more comma-seperated occurences of *X*.

Semantics for html related constructs

Constant

exp **::=** **<html>** *htmlbody_list* **</html>**

html constants are standard html documents augmented with named gaps and a few extra <bigwig> specific input fields (see Document Values above).

Requirements: An **html** constant must meet the gap requirements and field requirements mentioned above.

Variable

exp **::=** *id*

All variables in <bigwig> must be explicitly declared, **html** variables are no exception. As usual, they can be either local to the session or **shared**.

Requirements: **Shared html** variables are required to have the same flowtype throughout their existence. This flowtype is dictated by the flowtype of the initialization expression.

Assignment

exp **::=** *exp* **=** *exp*

The leftmost expression is evaluated first to an **html** l-value. Hereafter, the rightmost expression is evaluated to an **html** (r-) value which is assigned *by-value* to the location designated by the first.

Requirements: Both expressions must be of type **html**. Furthermore, the first must designate an l-value.

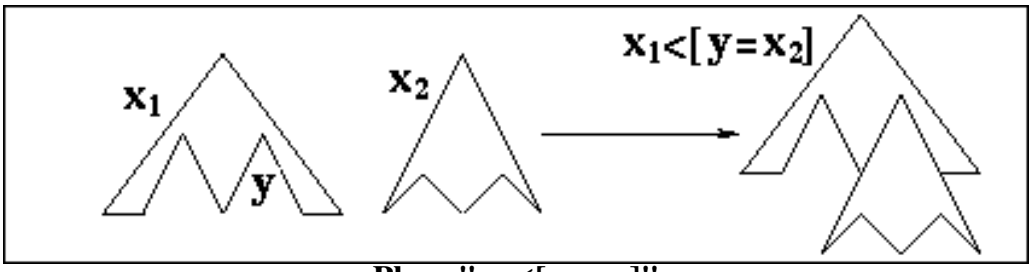
Plug

exp **::=** *exp* **<[** { *id* **=** *exp* }⊕ **]**

The first expression is evaluated to a document value that is required to have a gap by the name of the identifier. After this, the second is evaluated to a document value. Then the second document is plugged (inserted) into a copy of the first where the gap was, filling the gap. Due to efficient runtime representation of document values, the operation takes constant time only! Note that "**<[**" is one token.

There are actually two kinds of plug operations depending on whether the second expression is an html document or a string:

x_1 <[y = x_2] document plug (html plug)
x_1 <[y = s] attribute plug (string plug)



Plug: "x_1 <[y = x_2]"

Requirements:

- ***Gap present:*** In both cases there must be a gap named "y" present in x_1. In the first, there is an additional requirement that the gap "y" be a document gap (i.e. not an attribute gap).
- ***Consistent gap union:*** In order to meet the gap requirement above, it is required that the two sets of gaps "*gaps*(x_1) \ {y}" and "*gaps*(x_2)" are disjoint. This will be the set of gaps for the resulting document.
- ***Consistent field union:*** The fields of the two documents added together must meet the field requirements mentioned above.

Function Call

id **(** { *exp* }⊛ **)**

Functions may take html documents as parameters (which are treated as *call-by-value*) and return html documents. However, since we use a monovariant interprocedural dataflow analysis to infer document flow types, there are certain demands on the functions. All free html variables in a function and all actual parameters must have the same flow types at all calling points for the function. Also, the flow type of the returned document must be the same for all calling points.

Track

track (*exp*)

This built-in function is purely included to aid debugging. It acts as the identity function on html documents, but with one important *compile-time* side-effect. At *compile-time*, all **track** expressions will report the flow type of their argument as inferred by the flow analyzer.

Flash

stm **::= flash** *exp* **;**

The expression, that must be of type **html**, is evaluated yielding a document value, whose remaining gaps are implicitly closed (filled with nothing). The effect of executing the **flash** statement is that whenever no response is given within some time bound (eight seconds by default), this document appears in the client's browser. It can be used to give a reason explaining the delay (e.g. *"Searching database - this might take a while..."*). By default, a

document stating *"Reply not ready yet - please wait..."* is flashed. This may be overwritten by any other flash statement.

Requirements: The expression is of type **html** and the flashed document must not contain any input fields.

Show

stm **::= show** *exp* **;**

The expression, that must be of type **html**, is evaluated yielding a document value, whose remaining gaps are implicitly closed (filled with nothing). If no continue fields were present, a default **continue** button labeled "Continue" (or a submit button if the service is compiled with "--nojavascript") will be added to the document (lefthand bottom corner). The document is subsequently shown to the client after which the session goes to sleep. When the client clicks one of the continue fields, the session resumes from whence it paused (with the local state preserved).

Requirements: The expression is of type **html** and the shown document must not contain any value-submitting input fields.

Show-Receive

stm **::= show** *exp* **receive [** { *exp* **=** *id* }⊛ **] ;**

The semantics is as for **show**. When the document is submitted, the input fields listed in the receive are assigned to the mentioned variables. After this, execution resumes (with the local state preserved).

Requirements: The expression must be of type **html** and the number of received program variables must equal the number of document input fields in the document. Also, the types of the received program variables must match the corresponding input field kinds in the document. No input field or program variable may be mentioned twice in the receive assignment list.

Show/Timeout

stm **::= show** *exp* **timeout** *stm*

stm **::= show** *exp* **receive [** { *exp* **=** *id* }⊛ **] timeout** *stm*

The **show** statement may include a **timeout** statement that will be executed if the client does not submit the page shown before a certain amount of time has passed. This amount of time can be defined either globally in the <bigwig> configuration file (SPANDEFAULT [48hrs by default]) or per service, per session, or even per **show** statement using the **span** modifier. The **span** modifier takes an expression in parentheses and can be placed after the **show** keyword in **show** statements.

Exit

stm **::= exit** *exp* **;**

The expression, that must be of type **html**, is evaluated yielding a document value, whose remaining gaps are implicitly closed (filled with nothing). The **exit** statement will show the computed document to the client and the service will terminate. Exit can also be called with a string (designating a URL) argument or with no arguments. We refer to the Control Structures

section for information on these two variants.

Requirements: The expression is of type **html** and the exited document must not contain any input fields.

Flow Join Requirements

Because the flow analysis needs to infer the flow types of all html variables at all program points, all flow paths joining together must agree on the flow types of all live document variables.

If a gap is not present in all flows joining together at some program point, the gap is *implicitly closed* (plugged with nothing) by the compiler and henceforth no longer available for plugging.

These rules may appear slightly complicated and may take a little getting used to, but they are crucial in ensuring static safety.

HTML prototypes

html ::= <**html**> *htmlbody_list* </**html**> @ *stringconst*

The *stringconst* must designate an html file. If the file is *not* present (at *compile-time*), the value of the construct is the value of the constant document listed. If, however, the file *is* present (at *compile-time*), it will be the value of the construct. The html document in the file is required to have the same document type as the constant document (i.e. same gaps and fields) which is checked at *compile-time*. The idea behind this construct is to provide a means for rapid prototyping. The programmer rapidly makes some *prototype* html documents, with focus on the functionality (fields and gaps) and not on the graphical layout of the document. Then, as the *real* documents are gradually created, they replace the prototype ones.

Code Gaps

Code gaps are reminiscent of PHP and ASP evaluation tags. There are two kinds available in <bigwig>:

Code Expressions:

htmlbody ::= <[(*exp*)]>
attr ::= [(*exp*)]

Documents are allowed to contain in-lined expressions that are evaluated just before the document is shown. The value of such an expression is coerced to a string and inserted in the document in the place of the code expression.

Code Statements:

htmlbody ::= <[*compound_stm*]>
attr ::= [*compound_stm*]

The same thing is possible with statements. The last statement in the compound_stm must be a

statement-expression (*stm* ::= *exp* ;) whose type must not be void and whose value (coerced to a string) is inserted in the document.

Code gaps are evaluated in the order of occurrence in the document shown.

More information:

Research paper: "*A Type System for Dynamic Web Documents*".

<Database>

[Shared data | Schema Declarations | tuple | relation | vector]

Shared data

All variables declarations preceeded by the modifier shared will be persistent and *shared* among all session threads and will be managed by <bigwig>'s internal database. Fine grained concurrency control is not provided by the compiler and must be programmed explicitly by the service programmer. However, this happens transparently when using the high-level concurrency abstractions supplied by the standard macros library ("std.wigmac").

The macro package "sql.wigmac" provides an SQL like interface to the database.

Schema Declarations

All tuples, relations, and vector variables need to be declared according to some schema. The schemas themselves are declared explicitly as follows:

schema ::= **schema** *id* { *field_list* }
field_list ::= *field**
field ::= *type* *id_list* **;**

Note that schemas are only allowed to contain basic types.

Tuples

The **tuple** type is defined relative to a **schema** which is essentially a mapping of identifiers to basic types. A tuple value is thus a mapping from identifiers to basic types corresponding to the names and types of the associated **schema**. The tuple identifiers are commonly referred to as attributes. The following constructor is available for creating **tuple**s:

exp ::= **tuple** { *exp_list* }

The expression list must be assignments to identifiers that are the attribute names. Below is a list of the operators for manipulating **tuple**s.

Operator	Description
___==___	**equal.** The conjunction of the equality of the tuple's constituents.
___!=___	**not equal.** The negation of the conjunction of the equality of the tuple's constituents.
___=___	**assignment.** Conceptually *assign-by-value*, but implemented as *copy-on-write*.

__<<__	**left tuple overwrite.** Creates a tuple with the union of all the attributes of the two tuples. Whenever both tuples have a given attribute, the value of the rightmost argument tuple is taken.
__>>__	**right tuple overwrite.** Creates a tuple with the union of all the attributes of the two tuples. Whenever both tuples have a given attribute, the value of the leftmost argument tuple is taken.
__.__	**tuple member reference.** Accesses a member field in a tuple.
__\+__	**tuple project.** Creates a copy of the left argument tuple, keepin only the attributes mentioned in the right argument (a comma separated list of identifiers in parentheses).
__\-__	**tuple project complement.** Creates a copy of the left argument tuple, projecting away all attributes mentioned in the right argument (a comma separated list of identifiers in parentheses).

Relations

A relation is a set of tuples with no notion of order and where no value is present twice (any redundant values are implicitly removed). The following two operators operate on relations:

Operator	Description
__=__	**assignment.** Conceptually *assign-by-value*, but implemented as *copy-on-write*.
|__|	**size.** Returns the number of tuples in the relation.

The following built-in functions operate on relations:

exp ::= **relation** { *exp_list* }

Takes a comma separated list of tuple expressions in curly brackets and turns them into a relation.

exp ::= **cart** (*exp* , *exp*)

Takes two relation arguments in parentheses and produces the cartesian product.

exp ::= **factor** (*exp_list*) *stm*

exp ::= **factor** (*exp_list* ; *id_list*) *stm*

First we explain the semantics of the simplest possible factor expression, namely when there is only one expression and no identifiers. The expression is evaluated to a relation on which the factor expression will operate. The statement is executed once for each tuple in this relation where "#" will hold the current tuple. The statement may return tuples or relations (all required to have the same schema) which are union'ed together to form the ultimate result of the factor operation.

The next case is when a comma separated list of identifiers is supplied after the semi-colon. The relation resulting from the evaluation of the expression argument is projected onto these attributes 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. The remaining attributes will be available in the variable "@" (or "@1") that will be a relation containing all tuples that contributed to the current "#" tuple.

Finally, multiple expressions may be given as arguments. In this case, all expressions are

evaluated to relations from left to right. If no identifiers are specified, the type of "#" will be a tuple with the intersection of the attributes from all the expressions. As usual, the statement will be executed on the relation with all such tuples, setting "#" to the current. The individual contributions from the expressions will be available in the variables "@1" to "@*n*", where "*n*" is the number of identifiers specified ("@" is a shortcut for "@1").

Vectors

There are two kinds of Vectors (a.k.a. lists or arrays) in <bigwig>; basic vectors and tuple vectors. Basic vectors are capable of storing lists of basic values, while tuple vectors contain lists of tuples. Both kinds can be multi-dimensional. The following operators operate on vectors:

Operator	Description
`__=__`	**assignment.** Conceptually *assign-by-value*, but implemented as *copy-on-write*.
`__+__`	**concatenation.** Returns the concatenation of two vectors.
`__[__]`	**index.** Takes a vector and an integer (`n`) in square parentheses and extracts the nth element from the vector.
`__[__..__]`	**range.** Takes a vector and two integers (`low` and `high`) separated by two dots and extracts the subvector beginning with `low` and ending in `high`-1.
`|__|`	**length.** Returns the number of elements in the vector.

The following built-in functions operate on vectors:

exp ::= **vector** { *exp_list* }

Takes a comma separated list of expressions (of the same type) in curly brackets and constructs a vector from them.

exp ::= **sort** (*exp*)

exp ::= **sort** (*exp* **;** *id_list*)

There are two variants of the **sort** function; one for sorting basic vectors and one for sorting tuple vectors. The basic vector variant takes a basic vector and sorts it. The tuple vector variant takes a tuple vector, a semi-colon token, and a list of identifiers and sorts the tuple vector according to the attributes specified in the identifier list.

<Security>

[htaccess | md5 | selective | singular | ssl]

The keywords below are all modifiers that affect the level of security of certain areas in the service. They can be applied to an entire **service**, a **session**, a **show**, a **flash**, or an **exit** statement.

htaccess

Takes as argument a string designating a file that is to be used for login-password verification through the browser/web-server built-in htaccess protocol.

md5

This built-in (**m**essage **d**igest 5) function, md5, takes a string and returns a string, namely the md5 hash value of the argument given.

selective

This modifier takes a comma separated list of string expressions in parentheses as arguments and verifies that the IP number of the client is in this list. There are three possibilities for the strings in the list:

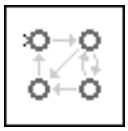
- *A (partial) domain-name*: Host whose name is, or ends in, this string are allowed access.
- *A full IP address*: An IP address of a host allowed access
- *A partial IP address*: The first 1 to 3 bytes of an IP address, for subnet restriction.

singular

This modifier will cause the server to inspect the IP number of the client when the session is started and subsequently verify that it does not change. Thus, it makes sure the client remains the same throughout the execution of the session.

ssl

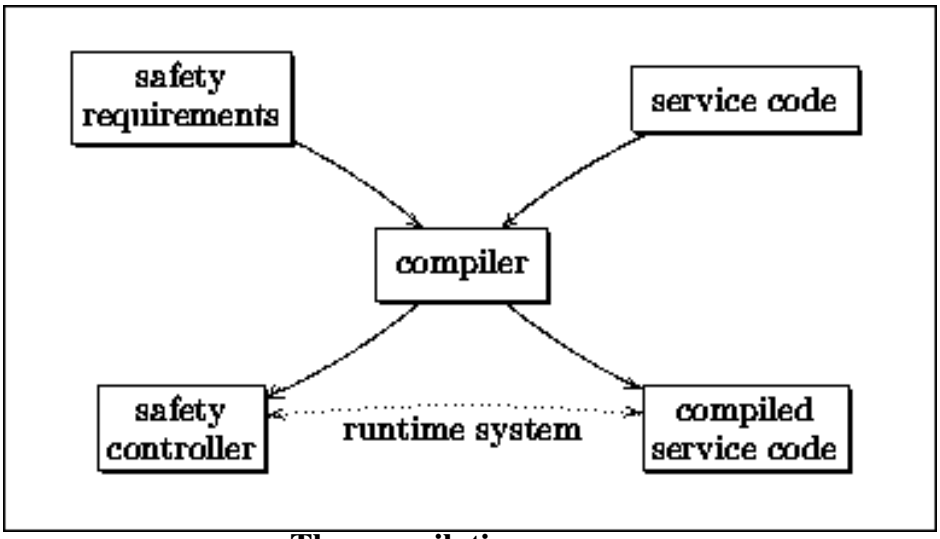
This modifier will make the entire service or a particular session subject to the SSL cryptographic protocol. It requires that the web server supports SSL.



<Concurrency Control>

[The Compilation Process | Constraints | Labels | Formulas | Triggers | Wait]

The Compilation Process



The compilation process.

The <bigwig> service code can contain checkpoints (**wait**) statements. The global sequence in which individual session threads at runtime pass these checkpoints may be constrained by formulas specified in monadic first-order logic (on strings/sequences). The <bigwig> service program is compiled into a C/CGI script and the logic formulas into a runtime safety controller (based on deterministic finite automata - DFAs). Every time a session thread (executing the compiled service code) wants to pass a checkpoint (at runtime), the safety controller is asked for permission. If the entire system is in a state in which the passing of this checkpoint does not violate the specified constraints, permission is granted, otherwise the session thread waits until it is safe to proceed.

Constraints

Constraints along with the **wait** statements control concurrency. The **wait** statement provides a means for inserting labels (or *checkpoints*) in the code for the session to pass as it executes. At any time, a service will have a number of executing sessions. Together, these sessions will give rise to a perpetually growing sequence of labels, namely the sequence of labels past in **wait** statements. It is this sequence that can be constrained through the use of contraints. Unwanted execution sequences may be described in logic and prohibited from occurring. All contraints are compiled into one centralized runtime controller process that will make sure the execution of sessions never violates the specified constraints.

Constraints are declared at toplevel and are comprised of three elements: label events, logic formulas, and trigger events.

The macro package "std.wigmac" provides a number of high-level concurrency control abstractions.

Labels

Labels need to be explicitly declared. They are what is passed in the **wait** statements and what is referred to in the formulas.

Formulas

The formulas are specified in monadic first-order logic on strings (here, the sequence of labels past in **wait** statements during execution).

Constants

The two constants formulas **true** and **false** are available.

Operators

The table below holds all the operators available in the formulas. The precedences are listed in *decreasing* precedence order. For instance, logical OR binds stronger than logical implication, which means that "`true => false || true`" is understood as "`true => (false || true)`"

Operator	Description	Operands	Assoc.
`(___)`	parentheses	formula	N/A
`!___`	negation	formula	N/A
`___&&___`	logical AND	formulas	left
`___||___`	logical OR	formulas	left
`___=>___`	logical implication	formulas	right
`___<=>___`	logical bi-implication	formulas	right
`___==___`	equal	q.v.	N/A
`___!=___`	not equal	q.v.	N/A
`___<___`	less than	q.v.	N/A
`___>___`	greater than	q.v.	N/A
`___<=___`	less than or equal	q.v.	N/A
`___>=___`	greater than or equal	q.v.	N/A

q.v.

Refers to quantifier variables.

left

means that the given operator is **left-associative**. For instance, "`x && y && z`" will be parsed as "`(x && y) && z`".

right

means that the given operator is **right-associative**. For instance, "`x => y => z`" will be parsed as "`x => (y => z)`".

Quantifiers

There are two quantifiers, namely universal and existential quantification. The syntax of the two constructs is:

all *id* **:** *formula*
is *id* **:** *formula*

The contructs both introduce a quantifier variable which is available in the subsequent formula. The quantifier variable refers to points in the global sequence of labels past in **wait** statements during execution.

Restrict-by

The formula restrict-by has the following syntax:

restrict *formula* **by** *id*

It is true if and only if the *formula* is true where all its quantifier variables are restricted to be before the point indicated by the identifier (quantifier variable).

Triggers

Triggers are a means of getting beyond the boundaries of regularity imposed by the logic. A trigger is declared as follows:

trigger : **trigger** *id* **when** *trigexp* == *trigexp* **;**
trigexp : *trigexp* + *trigexp*
| *trigexp* − *trigexp*
| + *trigexp*
| − *trigexp*
| *intconst*
| # *id*

A trigger declares an event (named by the identifier) that is triggered once each time the two trigger expressions are equal (goes from unequal to equal). Trigger expressions are integer expressions, counting the number of times various labels are past by **wait** statements. They can be referred to in the formulas as with labels and in this way provide the counting facilities required to take us beyond regularity.

Wait

```
stm ::= wait id ;
stm ::= wait { waitbranch-list }
waitbranch-list ::= waitbranch+
waitbranch ::= case id : stm-list break;
waitbranch ::= timeout exp : stm-list break;
```

The **wait** statement is special to <bigwig>. It provides the interaction with the *runtime controller*. There are two kinds of **wait** statements, namely the *short wait* corresponding to the first two

productions above, and the *general wait* corresponding to the third production above.

The short wait:
When executing a short wait statement "**wait** L;" (where `L` is some label defined as a constraint), the session thread will contact the runtime controller, requesting permission to pass the label `L`. If the entire service is in a state where it is *safe* for the session thread to pass the label `L` (i.e. without violating the safety contraints, meaning that `L` is *enabled*), the runtime controller will grant the session thread permission to proceed. Otherwise the session is suspended (even indefinitely), until it is safe for it to proceed.

The general wait:
The general wait is like the short wait, except that it allows the session thread to branch on the state of the runtime controller. The semantics is that permission to proceed is granted when *one* of the labels is passed (enabled). Execution proceeds afterwards at the **`case`** corresponding to the label passed. A general wait can also contain one **`timeout`** branch (where the expression evaluates to an integer). The expression is evaluated to a time bound (in seconds) and if the runtime controller cannot grant permission to continue within this bound, execution resumes with the timeout statement-list. Only one **`timeout`** is allowed in a **wait** statement and it must be specified after all the **`case`**s.

<Web Specifics>

[get | post | match | refresh | sendmail | url | dir | userid]

Get and Post

exp : := **get** (*exp*)

exp : := **get** (*exp*) [*exp_list*]

The expression is evaluated to a string designating a URL. The expressions in the expression list must all be assignments. The left hand sides in the assignments will be the `names` and the right hand sides, the `values`. This URL is fetched dynamically from the internet, supplying the string arguments specified in the expression list using the GET method. The result is a **string** containing the internet document fetched (it cannot be of type **html** since we do not statically know its flow type).

exp : := **post** (*exp*)

exp : := **post** (*exp*) [*exp_list*]

The expression is evaluated to a string designating a URL. The expressions in the expression list must all be assignments. The left hand sides in the assignments will be the `names` and the right hand sides, the `values`. This URL is fetched dynamically from the internet, supplying the string arguments specified in the expression list using the POST method. The result is a **string** containing the internet document fetched (it cannot be of type **html** since we do not statically know its flow type).

Match

exp : := **match** (*exp* , *exp*) [*exp_list*]

There are two variants of the **match** construct. The first argument must always be a string, namely the string on which the match construct will operate. The second argument can either be an html document template with a number of gaps or a format with a number of record constructs. The expressions in the expression list must all be assignments. The right hand sides will denote gaps or record constructs and the left hand sides will denote variables that will be assigned the values matched in the gaps or record constructs.

Matching against html document templates:

The string is matched against the html document template, requiring the non-gap constituents (text and fields) to match completely and treating the gaps as "wildcards" that will match anything. The construct employes an eager strategy to find a match and if successful, the assignments in the expression list will take place. The expressions on the right hand sides of the assignments must be identifiers naming the gaps of the document, evaluating to the parts of the string they matched. The string typically comes from a get or a post operation and can thus be used to extract certain patterns in a document gotten from the internet. The construct will

return a boolean signalling whether a match was possible or not.

Matching against regular expression formats:

The string is matched against the regular expression format, any record constructs will record what their internal regular expressions matched. The expressions on the right hand sides of the assignments must be identifiers naming the recording constructs in the format, evaluating to the parts of the string they matched. The construct will return a boolean signalling whether a match was possible or not.

Miscellaneous

mode **: : =** **refresh** **(** *exp* **)**

This construct sets the auto refresh rate used when documents are flashed, using the flash statement. The expression must evaluate to an integer which becomes the new refresh rate in seconds. The area of effect is either an entire service, a session, or a single flash statement, depending on where the *mode* was applied.

exp **: : =** **sendmail** **(** *exp* **,** *exp* **)**

This construct sends an email. The first argument must be a relation or a vector with a schema with two attributes named "name" and "value". The second argument is a string containing the actual message to be sent. The relation/vector part will be used to generate the header information in the email and should for this reason at least contain "to", "subject" and "from" tuples with corresponding "value"'s.

exp **: : =** **url**

exp **: : =** **url** *id* **(** *exp_list* **)**

The **url** expression (without arguments) returns a string which is the URL of the HTML reply file associated with the current session. The variant with arguments will take an identifier naming a session in the current service plus a list of appropriate arguments. The construct will not call the session with the arguments, but instead provide an html string that does (using the GET method) when accessed by a browser.

exp **: : =** **dir**

The **dir** expression returns a string which is the directory of the HTML reply file associated with the current session. It is typically used for manipulating files in the session's private directory.

exp **: : =** **userid**

This expression allows a service to remember the identity of a client beyond a session run through the use of a cookie. If the expression is present somewhere in a service, the service will when started generate a random string which is set as a cookie by the runtime system. The expression **userid** will henceforth return the value of this string.

exp **: : =** **getcookie** **(** *exp* **)**

This built-in function takes a **string** argument naming a cookie and returns the value of this cookie. The function returns the empty string if there is no cookie with the given name.

exp **: : =** **setcookie** **(** *exp* **,** *exp* **)**

This built-in function takes two **string** arguments. The cookie named by the first argument is

set to the value designated by the second argument. However, the cookie setting will only take effect at the next **show** or **exit** statement.

<Time>

[Values | now | notime | Type Constructors | Get | difftime | Operators]

Values

The values of type `time` are legal points in time. That is, a (gregorian calender) date and an hour-minute-second time of day. Illegal times, such as "`1999/02/29, 20:61:30`" are automatically converted to legal ones ("`1999/03/01, 21:01:30`"). All **time** values must be within the following range [`1970/01/01, 00:00:00..2038/01/19, 03:14:07`]. Any attempts to construct a **time** not in this interval will result in the special **time** value called **notime** which is also the default value for the type **time**. All **time** values are assumed to be in the server's local timezone.

---

now

This predefined function returns the current local time.

---

notime

This is a constant time value which is less than all other time values. It is the default value for variables of type time and it is produced whenever illegal time operations are performed. Attempts to get components from a notime value will produce a runtime error.

Type Constructors

exp : := **time** (*exp* , *exp* , *exp*)

This time contructor takes three integer arguments, a year (1970-2038), a month (1-12), and a day (1-31) and will construct the corresponding point in time (with time-of-day: 00:00:00).

exp : := **time** (*exp* , *exp* , *exp* , *exp* , *exp* , *exp*)

This time contructor takes six integer arguments, a year (1970-2038), a month (1-12), a day (1-31), an hour (0-23), a minute (0-59), and a second (0-59) and will construct the corresponding point in time.

Get

exp : := **getYear** (*exp*)

Returns an integer (1970-2038) with the year component of the given time value.

exp : := **getMonth** (*exp*)

Returns an integer (1-12) with the month component of the given time value.

exp : := **getDay** (*exp*)

Returns an integer (1-31) with the day component of the given time value.

exp : := **getHour** (*exp*)

Returns an integer (0-23) with the hour component of the given time value.

exp : := **getMinute** (*exp*)

Returns an integer (0-59) with the minute component of the given time value.

exp : := **getSecond** (*exp*)

Returns an integer (0-59) with the second component of the given time value.

exp : := **getWeekday** (*exp*)

Returns an integer (1-7) with the weekday component of the given time value (Monday is one and Sunday is seven).

---

difftime

exp : := **difftime** (*exp* , *exp*)

This predefined function takes two time values and produces an integer holding the difference between the two points in time in seconds.

Operators

The following binary operators are available on values of type time:

Operator	Description
___<___	less than
___>___	greater than
___<=___	less than or equal
___>=___	greater than or equal
___==___	equal
___!=___	not equal
___=___	assignment

Recent BRICS Notes Series Publications

NS-00-4 Claus Brabrand. *<bigwig> Version 1.3 — Reference Manual*. September 2000. ii+56 pp.

NS-00-3 Patrick Cousot, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raussen, and Vladimiro Sassone, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '00,* (State College, USA, August 21, 2000), August 2000. vi+116 pp.

NS-00-2 Luca Aceto and Björn Victor, editors. *Preliminary Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00,* (State College, USA, August 21, 2000), August 2000. vi+130 pp.

NS-00-1 Bernd Gärtner. *Randomization and Abstraction — Useful Tools for Optimization*. February 2000. 106 pp.

NS-99-3 Peter D. Mosses and David A. Watt, editors. *Proceedings of the Second International Workshop on Action Semantics, AS '99,* (Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.

NS-99-2 Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99,* (Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.

NS-99-1 Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,* (San Antonio, Texas, USA, January 22–23, 1999), January 1999.

NS-98-8 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,* (Gothenburg, Sweden, May 8–9, 1998), December 1998.

NS-98-7 John Power. *2-Categories*. August 1998. 18 pp.

NS-98-6 Carsten Butz, Ulrich Kohlenbach, Søren Riis, and Glynn Winskel, editors. *Abstracts of the Workshop on Proof Theory and Complexity, PTAC '98,* (Aarhus, Denmark, August 3–7, 1998), July 1998. vi+16 pp.