Compaq's Web Language
A Programming Language for the Web

Hannes Marais

Compaq Systems Research Center (SRC)

This document describes version 3.0 of Compaq's Web Language, hereinafter abbreviated to WebL, its former name. See also: *http://www.compaq.com/WebL*

Acknowledgements: WebL was initially designed and implemented by Thomas Kistler and Hannes Marais. Service combinators were contributed by Luca Cardelli and Rowan Davies. Tom Rodeheffer suggested many improvements to the language and implementation. Monika Henzinger, Jeff Dean, Brian Eberman and Jin Yu contributed many suggestions, bug fixes, and improvements. Cynthia Hibbard, Dominique Marais, and Krishna Bharat corrected several mistakes in the user manual. Since the release of the software in July 1998, many corrections and improvements have been made by WebL users themselves. The list of contributors and their contributions are contained in the file *BugList.java,* which is part of the WebL source distribution.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

WebL (pronounced “webble”) is a web scripting language for processing documents on the World Wide web. It is well suited for retrieving documents from the web, extracting information from the retrieved documents, and manipulating the contents of documents. In contrast to other general purpose programming languages, WebL is specifically designed for automating tasks on the web. Not only does the WebL language have a built-in knowledge of web protocols like HTTP and FTP, but it also knows how to process documents in plain text, HTML and XML format.

The flexible handling of structured text markup as found in HTML and XML documents is an important feature of the language. In addition, WebL also supports features that simplify handling of communication failures, the exploitation of replicated documents on multiple web services for reliability, and performing multiple tasks in parallel. WebL also provides traditional imperative programming language features like objects, modules, closures, control structures, etc.

To give a better idea of how WebL can be applied for web task automation, and also what makes WebL different from other languages, it is instructive to discuss the computational model that underlies the language. In addition to conventional features you would expect from most languages, the WebL computation model is based on two new concepts, namely *service combinators* and *markup algebra*. For now we can describe these two concepts of WebL in the following way.

Service combinators is a formalism that can provide more reliable access to web resources and services. Very succinctly, service combinators is an *exception handling mechanism* that is powerful enough to encode robust behavior when communication failures occur. This concept is especially important for performing any reliable computation on the unreliable web structures. It often happens that web services are unavailable, suddenly fail or become unacceptably slow. These are very serious complications for computations that depend so much on the web infrastructure. Although service combinators cannot make a web-based computation completely failure-proof, it does add a certain amount of robustness to programming on the web. Service combinators are discussed in detail on page 47.

Markup algebra is a formalism for extracting information from structured text documents and the manipulation of those documents. It consists of functions to extract elements and patterns from web documents, operators to manipulate what has been extracted in this manner, and functions to change a page, for example to insert or delete parts. The functions and operators all work on the high-level concept of a parsed web page, and there is little need to do lower level string manipulation. Markup algebra is discussed in detail in Chapter 4.

The purpose of this document is to introduce programmers to the WebL language and its features. Before however introducing the language in its totality, we will first summarize WebL's main features.

Basic Features

- The WebL language and system is designed for rapid prototyping of *Web computations*. It is well-suited for the automation of tasks on the WWW.
- WebL's emphasis is on high flexibility and high-level abstractions rather than raw computation speed. It is thus better suited as a rapid prototyping tool than a high-volume production tool.
- WebL is implemented as a stand-alone application that fetches and processes web pages according to programmed scripts.

Programming Language

- WebL is a high level, imperative, interpreted, dynamically typed, multi-threaded, expression, language.
- WebL's standard data types include boolean, character, integer (64-bit), double precision floats, Unicode strings, lists, sets, associative arrays (objects), functions, and methods.
- WebL has prototype-like objects.

- WebL supports fast immutable sets and lists.
- WebL has special data types for processing HTML/XML that include pages, pieces (for markup elements), piece sets, and tags.
- WebL uses conventional control structures like if-then-else, while-do, repeat-until, try-catch, etc.
- WebL has a clean, easy to read syntax with C-like expression and Modula-like control structures.
- WebL supports exception handling mechanisms (based on Cardelli & Davies' service combinators) like sequential combination, parallel execution, timeout, and retry. WebL can emulate arbitrary complex page fetching behaviors by combining services.

Protocols Spoken

- WebL speaks whatever protocols Java supports, i.e. HTTP, FTP, etc.
- WebL can easily fill in web-based forms and navigate between pages.
- WebL has HTTP cookie support.
- Programmers can define HTTP request headers and inspect response headers.
- Programmers can explicitly override mimetypes and DTDs used when parsing Web pages.
- Proxy support.
- Support for HTTP basic authentication (both client and proxy authentication).

Markup Algebra

- WebL 'understands' HTML, XML and plain text mime-types.
- WebL uses a DTD-based HTML parser for extensibility (HTML 2.0, 3.2, and 4.0 DTDs included).
- WebL has relatively robust page parsing that attempts to make a faithful representation of Web pages.
- WebL supports a markup algebra for extracting elements and text from pages, and functions for manipulating the content of a page. Extraction functions include extracting all elements of a specific name, all occurrences of PERL5 regular expressions, and all occurrences of simple element patterns.
- Elements and patterns are mapped onto piece objects in WebL, and allow the direct access to markup attributes.

- Markup algebra allows the expression of complicated access patterns easily (for example, "extract all the images in the third row of the table (that contains the word ”abc”"), and so on).
- WebL can handle overlapping elements internally. (Page manipulation is not based on an internal tree-like representation of markup.)
- Page manipulation functions include modifying attributes, deleting elements/ tags, copying elements/text, and replacing elements/text.
- WebL allows programmers to look at both the markup structure of a page and the raw text (without any tags).

Module Support

Standard modules supplied with WebL include:

- File manipulation for writing or downloading pages to disk.
- Displaying pages in your web browser, checking which pages are being viewed in Netscape, and instructing Netscape to navigate to a specific URL (Windows only).
- Multi-processing with workers, jobs, and job queues.
- General string manipulation including PERL5 regular expression searches.
- Routines to split and glue together URLs.
- An easily customizable multi-threaded web crawler.
- A multi-threaded web server that allows the direct execution of WebL functions with full access to HTTP state.
- Java servlet support.
- Examples to access information from public services like AltaVista, Yahoo!, etc.

Java Support and Integration

- WebL is written in nearly completely in Java. (The Browser access module needs access to a few Windows API calls; WebL is completely portable on UNIX platforms.)
- It is possible (however not recommeded) to directly code against the WebL API (thus not writing WebL scripts but still using its functionality).
- Very easy to add bridges from WebL to Java code. Java objects can be called directly from WebL code without extending the WebL system (see module *Java*).

- Java extensions are loaded dynamically and it is possible to add and remove builtin functions by editing a standard script.

Applications

WebL is a general purpose programming language, and can thus be used to build whatever you can imagine. The example chapter of this book only gives a small taste of what is possible with WebL. Some of the things that we at Compaq have built with WebL include:

- Web shopping robots,
- Page and site validators,
- Meta-search engines,
- Tools to extraction connectivity graphs from the Web and analyze them,
- Tools for collecting URLs, host names, word frequency lists, etc.,
- Page content analysis and statictics,
- Reprocessing of results from public services, for example custom rankings of stocks,
- Custom servers and proxy-like entities,
- Locating and downloading multi-media content,
- and downloading of complete Web sites.

CHAPTER 2

The Language Core

The special features of WebL, like service combinators and markup algebra, are integrated in a small *programming language core.* Stripped of special features, the core language is conceptually similar to most other procedural programming languages. To lay some ground work, and to understand the examples introduced in later chapters, we first need to study the language core without touching the special features the language introduces for handling web pages. This chapter introduces many of the essential and basic language concepts that you will need to know in later chapters.

Basic Terminology

Expressions

WebL programs consist of sequences of *expressions* separated by semicolons. *Running* a WebL program involves *evaluating* the expressions in sequence. Each expression either evaluates to a *value* (or *result*) or causes an *exception* that causes the program evaluation to terminate at that point. (We say that the expression *throws* an exception. More details about exceptions can be found in the sections "Try Statement" on page 36 and "Exceptions" on page 182.)

The value of one expression is typically used by other expressions in the program. We also define the value of a sequence of expressions to be the value that the *last* expression in the sequence evaluated to. If no special steps are taken by the programmer, the results of the remainder of the single expressions in an expression sequence are lost.

Running or *executing* a WebL program involves several integrated steps:[1]

- The program source text is parsed and checked for syntax errors. If syntax errors are detected, the execution of the program is terminated.
- A representation of the program in the form of an abstract syntax tree (AST) is constructed in memory. The AST consists of a sequence of expressions.
- The in-memory sequence of expressions are executed in turn. Side effects of the computation might be printing of results on the console, or communicating over the Internet. The value of the expression sequence (the last expression executed) is discarded.

Value Types

Each *value* has an associated *value type* or *type*. The type determines how the value can be further used by expressions. For example, it is only possible to multiply two values that have a numerical type. WebL is a dynamically typed programming language. This means that at the point where values are used by expressions, they are checked to be of the correct type as expected by the expression. If they are not, an exception is thrown.

The defined value types of WebL are: *nil*, *boolean*, *int*, *real*, *char*, *string*, *fun*, *meth*, *set*, *list*, *object*, *page*, *piece*, *pieceset*, *tag*. (See “Dynamic Types” on page 23 for more details.)

1. Note that WebL programs need not be compiled explicitly — compilation is performed automatically just before the program is executed. WebL’s execution model is similar to that of most *scripting languages*.

Constants

Constants are simple expressions that evaluate to themselves. They are the simplest WebL expressions. WebL allows *nil*, *boolean*, *integer*, *real*, *character*, and *string* constants. Table 1 lists examples of constant expressions, what they evaluate to, and the resulting value type.

TABLE 1. Constant examples

Expression	Value	Value Type
nil	nil	nil
true	true	bool
false	false	bool
21	21	int
1.41	1.41	real
'a'	'a'	char
"abc"	"abc"	string
`abc`[a]	"abc"	string

a. Back-quoted string constants differ from regular string constants in that escape sequences contained in the string are not expanded.

Operators

Operators combine expressions into more complicated expressions. Evaluating an expression involves evaluating the operands (constituent expressions), performing some computation on the resulting values, and returning a result. Examples include numerical, boolean and service combinator operators. The evaluation sequence of operands is typically left to right.

TABLE 2. Operator examples

Expression	Value	Value Type
true or false	true	bool
2 + 2	4	int
1.2 * 2	2.4	real
"abc" + "def"	"abcdef"	string

Statements

WebL uses typical imperative program language constructs like *while*, *if*, and *try* statements. These statements are expressions in WebL, which means that they also evaluate to a value (often to the value *nil*).

Examples:

```
while x > 0 do x = x - 1 end

if x > y then y else x end

if x = 1 then y = x * 2
elsif x = 2 then y = x * 7
else y = 1
end

every s in "Hello World" do
   PrintLn(s)
end

repeat x = x * 2 until x > y end
```

Variables and Scoping

A *context* is a set of variables and associated value bindings. Expression evaluation is performed in a context which specifies the values of the variables that appear in the expression. A context can be created in several ways. The most common way is by the programmer, who defines the variables and their values explicitly (in the current context) using variable declarations. After declaration, a variable can be assigned arbitrary values with the assignment expression "=".

Examples:

```
var x;               // Defines the variable x.
var a, b, c;         // Defines three variables a, b, c.
var name = "John";   // Defines a variable called name
                     // and assigns it a value.
var                  // Define and initialize several
   x = 1,            // variables.
   y = 2,
   z = 3 * x;
x = y * 2;           // Assignment expression.
```

A variable's value is set to *nil* when no initializer is specified. A variable must be declared before it is used for the first time, and should be declared only once in any given context. Variable declarations are expressions that evaluate to the value the variable is set to. Assignment expressions evaluate to the value that is assigned.

It is important to note that

```
var x = x + 1
```

is equivalent to

```
var x = nil;
x = x + 1
```

Both of these programs lead to a runtime exception because 1 and *nil* are not type compatible under the plus operator. This definition of variable declaration allows the introduction of self-recursive functions.

WebL uses *lexical scoping* for variables. This allows contexts to be nested in each other according to the syntactic structure of the program. Nested contexts are automatically created at points where sequences of statements can be used, for example inside *while*, *repeat*, and *if* statements. A fresh context can be created explicitly with the *begin* statement. A variable can be used in a specific context at all positions syntactically following the place where it was declared.

Variable resolution is done by searching for a binding from inner (nested) contexts to outer contexts. This allows variables in inner contexts to override variables with the same name in outer contexts. For example, in the following program the variable *sq* is visible only inside the body of the *while* statement, and the variable *i* is visible only inside the context defined by the *begin* statement:

```
var sum = 0;
Print("The sum of the squares between 0 and 100 is ");
begin
    var i = 0;
    while i <= 100 do
        var sq = i * i;
        sum = sum + sq;
        i = i + 1
    end;
end;
PrintLn(sum)
```

Constructors

WebL also supports *lists* of values, *sets* of values, and *objects* with *fields*. *Constructors* perform the creation of these types of values from simpler values. Table 3 shows that lists are constructed by square brackets, sets by curly braces, and objects by square brackets and a period token. More information about these value types is given in the section “Dynamic Types” on page 23. Note that constructors consist of sub-expressions that are evaluated during value construction.

TABLE 3. Constructor expressions

Expression	Value	Value Type
[1, 2, 2 + 1]	[1, 2, 3]	list
{ 1, 2, 1 + 1 }	{ 1, 2 }	set
[. a = 1+1, b=2 .]	[. a=2, b=2 .]	object

Dynamic Types

Recall that WebL values have a value type that determines to a large extent what can be done with the value (i.e. what operators can be applied to it). The following paragraphs will explain the characteristics of each value type and give examples.

Type Nil

The keyword “nil” denotes a special value that indicates that a variable has no value. Note that we refer both to the value and the value type as *nil*. Variables that are declared without an initial value are initialized to the *nil* value.

Type Bool

The lexical constants *true* and *false* evaluate to a value of type *bool* (short for boolean). Expressions containing operators that compare values (for example *equal* or *less than*) also evaluate to a boolean. Boolean expressions can be combined with logical *and* and logical *or* operators, and are evaluated in a short-circuited fashion.

TABLE 4. Boolean Expressions

Expression	Value	Value Type
true	true	bool
false	false	bool
true or false	true	bool
true and false	false	bool
1 == 1	true	bool
1 <= 1	true	bool
1 != 1	false	bool

Type Char

A lexical character constant evaluates to a value of type *char* (or character). Characters are enclosed with single quotes. The internal coding of characters is Unicode. Character expressions may contain escape sequences that denote special characters. Table 6 lists the escape sequences used in WebL.

TABLE 5. Character Expressions

Expression	Value	Value Type
'a'	'a'	char
'\n'	'\n'	char
'a' + 'b'	"ab"	string

TABLE 6. Escape Sequences

Escape	Description
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash
\xxx	Character of octal value xxx
\uxxxx	Character of hexadecimal value xxxx

Type String

A lexical string constant enclosed in double quotes evaluates to a value of type *string*. A string consists of a sequence of characters. The number of characters in a string is called its *size*. The empty string, denoted by "", contains no characters and has a size of zero. There is no limit to the string size. Strings may be wrap across several lines in WebL programs. Strings may also contain the escape sequences defined in Table 6. Note that escape sequences are not expanded in strings that are written with the back-quote character.

TABLE 7. String Expressions

Expression	Value	Value Type
"abc"	"abc"	string
`ab\nc`	"ab\nc"	string
"abc" + ’d’	"abcd"	string
Size("abc")	3	int

Type Int

A lexical integer constant evaluates to a value of type *int* (or integer). The internal representation of integers is 64-bit signed two’s-complement. Overflows or underflows during integer computations do not throw exceptions.

TABLE 8. Integer Expressions

Expression	Value	Value Type
1 + 2	3	int
2 * (1 - 1)	0	int
6 div 4	1	int
6 mod 4	2	int

Type Real

A lexical real constant evaluates to a value of type real. The internal representation of reals is 64-bit IEEE 754 floating-point. No exceptions are thrown in real math.

TABLE 9. Real Expressions

Expression	Value	Value Type
1.2	1.2	real
2 / 2	1.0	real
0 / 0	NaN	real
1 / 0	+Inf	real

Type List

The list constructor [] constructs values of type lists. All expressions between the square brackets are evaluated from left to right and the values inserted into the list in that sequence. The parallel list constructor [| |] evaluates the expression between the brackets in parallel (using multiple threads) instead of left to right. There is no restriction on the size of the list or the value types that it can contain.

TABLE 10. List Expressions

Expression	Value	Value Type
[1 + 1, 2, "a"]	[1, 2, "a"]	list
[1, 2] + [3]	[1, 2, 3]	list
First([1, 2])	1	int
Rest([1, 2, 3])	[2, 3]	list
[| 1 + 1, 2 * 2 |]	[2, 4]	list
Size([1, 2, 6])	3	int

Type Set

The set constructor { } constructs values of type set. All the expressions between the curly braces are evaluated and their values inserted in the set (if not already an element of the set). There is no ordering between the elements, no restrictions on the type of elements, and no restriction on the size of the set.

TABLE 11. Set Expressions

Expression	Value	Value Type
{ 1+ 1, 2, 3}	{2, 3}	set
{1, 2, 3} + {2, 4}	{1, 2, 3, 4}	set
{1, 2, 3} * {2, 4}	{2}	set
{1, 2, 3} - {2, 4}	{1, 3}	set
Size({1, 2, 3, 6})	4	int

Type Fun

The *fun* statement constructs values of type *fun* (or function). The format of the fun statement is of the following form:

```
fun (arg1, arg2, ...)
    ... StatementSequence ...
end
```

The identifiers in brackets are the *formal arguments* of the function. A function can be *applied* with the same number of *actual arguments* enclosed in parenthesis following the function constructor. The actual arguments are evaluated and assigned in a paired manner to the formal arguments. The resulting variable bindings form a new context in which the statement sequence of the function is executed. The value of the applied function is the value of the function statement sequence (executed in the new context). For example,

```
fun(x, y) x + y end (3, 4)
```

evaluates a function that sums two numbers with arguments x=3 and y=4. More typically, functions are constructed and then assigned to variables for later use. For example, the following program calculates the factorial of 10:

```
var fac = fun(n)
    if n == 1 then 1
    else n * fac(n - 1)
    end
end;

fac(10)
```

The function context created during function execution is nested in the context in which the function was initially created. This allows the construction of higher order functions and closures. As an example, we define a function that returns a function that adds a certain number to its argument:

```
var MakeAdder = fun(c)
    fun(x) x + c end
end;
var Add5 = MakeAdder(5);

Add5(10);        // Add 5 to 10
```

WebL requires the introduction of a new variable before its first use. This creates a problem when functions need to mutually refer to each other, because one function has not been introduced yet at the place where it is called. Fortunately, as functions are first class citizens in WebL, the problem can be overcome by declaring mutually recursive functions by introducing the function variable names and afterwards assigning them values:

```
var f, g;

f = fun() ... g() ... end;
g = fun() ... f() ... end;
```

Type Object

The object constructor [. .] constructs values of type *object*. Objects have *fields*, each field having a specific *field value*. Fields are typically used to store object variables, functions, and methods inside the object. The fields themselves may be of any value type, i.e. they are untyped.

Indexing an object with the field retrieves the value of that field. There are two common ways of indexing into object fields:

- The *o.x* notation denotes a field called *x* of object *o*.
- The *o[e]* notation evaluates *e* to a value *x*, and retrieves the field *x* of *a*. This effectively makes the object an associative array.

The expression *o.x* and *o["x"]* refers to the same field. Trying to access an object field that does not exist will throw an exception. A special assignment expression ":=" is used to insert new fields into an object. (If the field already existed, its previous value is overridden.) A builtin function called *DeleteField* allows the removal of a field from an object.

Examples:

```
[. .]                       // The empty object
[. x = 1, y = 1 + 1 .]      // Object with x & y field

var o = [. x = 1 .];
o.x                         // Field x of o
o["x"]                      // The same field again
o.y := "hello"              // Defines field y of o
o[1+2] := 42                // Defines field 3 of o
o[4-1]                      // Accesses field 3 of o
Size(ToList(o))             // # fields of o
DeleteField(o, "x")         // Remove field "x"
```

The associative array behavior is so useful that it is used for other WebL types too. These object-like types are called *special objects*. Examples include types *page* and *piece*. To the programmer these types look very similar to objects, but they have "hidden" state attached to them (i.e. they function as opaque data types).

Object fields are ordered in the sequence of their definition (i.e. left-to-right, top-to-bottom in the object). The ordering of fields only has little impact in programs; it only defines the sequence in which fields are enumerated and how objects are printed. It does however play an important role for certain functions where parame-

ter ordering is important. (See “Retrieving Page Objects” on page 59.) Note that there is no way to remove a field from an object.

Object-based programming in WebL. Combining objects and functions allows us to program in an object-based or object-oriented manner. For example, the following program implements a bank account object with “methods” to deposit and withdraw money. Note how we need to pass the bank account object as first actual argument to the *deposit* and *withdraw* methods. In both cases the *self* formal argument refers to the bank account object.

```
var myaccount = [.
    balance = 0,
    deposit = fun(self, amount)
        self.balance = self.balance + amount
    end,
    withdraw = fun(self, amount)
        self.balance = self.balance - amount
    end
.]
myaccount.deposit(myaccount, 100);// Deposit $100
myaccount.withdraw(myaccount, 50);// Withdraw $50
```

Type Meth

The meth constructor constructs values of type *meth* (or method). Methods behave in all aspects, except for application (i.e. execution), in the same manner as functions. They are in fact used as a notational short-hand for method invocation without the need to pass a self parameter. We can recode the bank account program with methods in the following manner:

```
var myaccount = [.
    balance = 0,
    deposit = meth(self, amount)
        self.balance = self.balance + amount
    end,
    withdraw = meth(self, amount)
        self.balance = self.balance - amount
    end
.]
myaccount.deposit(100);       // Deposit $100
myaccount.withdraw(50);       // Withdraw $50
```

As can be seen, the only difference from the previous program is the use of the *meth* keyword, and a convenient way of invoking methods. In fact, the internal implementation of methods is equivalent to the bank account object programmed only with functions.

Types Page, Piece, PieceSet and Tag

Value types *page*, *piece*, *pieceset* and *tag* are an essential part of the WebL *markup algebra*. We will not go into details yet about these value types — they are discussed in more detail in Chapter 4.

Value Equality

Values of types *nil*, *boolean*, *int*, *real*, *char*, *string*, *fun*, *meth*, *set*, *list*, and *tag* are *immutable*. This means that once a particular value is calculated or declared in a constant, the value cannot change. For example, appending a character to a string creates a new string; inserting an element into a set creates a new set, and so on.

In contrast, objects and special objects are mutable by the fact that their field values can be modified, and new fields can be added.

Two immutable values are equal if their contents:

- are both *nil*.
- have the same boolean value (*true* or *false*).
- have the same numerical value (by converting ints to reals if necessary).
- have the same character or string.
- have identical sets or lists.
- have the same function or method with identical dynamic outer context.

Two objects are regarded equal when the internal reference to the object data stucture is equal (i.e. reference equality).

It is important to note that even though sets and lists are immutable in WebL, operating on these value types does not necessarily mean that the internal data structures are copied for each operation. WebL uses an efficient internal implementation that makes the following operations possible in constant time and space:

- Concatenating two lists,
- Applying the *First* and *Rest* functions on a list,
- Adding or removing an element from a set.

In some cases, for example when printing a list or indexing into it, a cost proportional to the number of elements is paid once, after which the cost becomes constant again.

Operators

Table 12 lists the operatorsof the WebL core language. To illustrate how operators are overloaded, we use a functional notation even though the operators are written in infix, prefix, or right-bracket fix. For example,

```
op(x: T, y: S): U
```

denotes that an infix operator *op* takes a first operand of *x* of type *T* and a second operand *y* of type *S*, and returns a value of type *U*. Unary operators have only a single argument to specify.

Two special operators are not contained in the operator table, since they have special constraints on when they can be used and hence cannot be specified in the syntax just introduced. The two operators are assignment ("=") and field definition (":=").

The left-hand side of an assignment must denote a variable or an object and field name combination. The left-hand side of a field definition must denote an object and field name combination, eg. *obj[field]* or *obj.field*. The value of an assignment or field definition is always the right-hand side of the operator. These two operators also differ in another way from the remainder of the operators, in that they have side-effects, namely the setting of the value of a variable or field of an object to the right-hand side of the operator.

TABLE 12. WebL Core Operators

Operator	Description
+(x: int, y: int): int +(x: int, y: real): real +(x: real, y: int): real +(x: real, y: real): real	Numeric addition x + y.
+(x: char, y: string): string +(x: char, y: char): string +(x: string, y: string): string +(x: string, y: char): string	String and character concatenation.
+(x: set, y: set): set	Set union.
+(x: list, y: list): list	List concatenation.
-(x: int, y: int): int -(x: int, y: real): real -(x: real, y: int): real -(x: real, y: real): real	Numeric substraction.
-(x: int): int -(x: real): real	Numeric negation.
-(x: set, y: set): set	Set exclusion.
*(x: int, y: int): real *(x: int, y: real): real *(x: real, y: int): real *(x: real, y: real): real	Numeric multiplication.
*(x: set, y: set): set	Set intersection.
/(x: int, y: int): int /(x: int, y: real): real /(x: real, y: int): real /(x: real, y: real): real	Numeric division.
div(x: int, y: int): int	Whole division.
mod(x: int, y: int): int	*x* mod *y*.
C(x: int, y: int): bool C(x: int, y: real): bool C(x: real, y: int): bool C(x: real, y: real): bool	Numerical comparison, where *C* is one of <, <=, >, or >=.
C(x: string, y: string): bool C(x: char, y: char): bool	Lexical comparison, where *C* is one of <, <=, >, or >=.

TABLE 12. WebL Core Operators

Operator	Description
== (x, y): bool	Value equality test. See “Value Equality” on page 31.
!=(x, y): bool	Value in-equality test. See “Value Equality” on page 31.
or(x: bool, y: bool): bool **and**(x: bool, y: bool): bool	Logical operators (Short-circuit evaluation).
!(x: bool): bool	Logical negation.
.(x: object, y): any	Object field access.
[](x: list, i: int): any [](x: object, i): any [](x: string, i: int): char	List, object, and string indexing[a]. Elements in a list and string are numbered from *0* to *Size-1*.
member(x, s: set): bool **member**(x, l: list): bool **member**(x, o: object): bool	Set, list and object[b] membership test.

a. Operator is written in the form *x[i]*.

b. Object membership test is based on object field names.

Statements

Statement Sequences

Statement sequences are separated by semicolons. The value of a statement sequence is the value of the last expression in the sequence. An optional trailing semicolon in a statement sequence is ignored by WebL.

If Statement

If statements specify the conditional execution of guarded commands. The boolean expression preceding an expression is called its *guard*. The guards are evaluated in sequence of occurrence until one evaluates to true, whereafter its associated expression is evaluated. If no guard is satisfied, the statement sequence following the symbol *else* is executed, if there is one. The value of an if statement is the value of the associated expression whose guard evaluated to true.

Syntax:

IfStat = **if** SS **then** SS [ElseStat] **end**
ElseStat = **else** SS | **elsif** SS then SS [ElseStat]

Example:

```
if ch >= 'a' and ch <= 'z' then ReadIdent()
elsif ch >= '0' and ch <= '9' then ReadNumber()
elsif ch == '\'' or ch == '"' then ReadString()
else ReadSpecial()
end
```

While Statement

While statements specify the repeated execution of an expression while a boolean expression (its guard) yields true. The guard is checked before every execution of the expression. The value of a while statement is *nil*.

Syntax:

WhileStat = **while** SS **do** SS **end**

Example:

```
while x > 0 do x = x div 2; k = k + 1 end
```

Repeat Statement

Repeat statements specify the repeated execution of an expression until a boolean expression (its guard) yields true. The guard is checked after every execution of the expression. The value of a repeat statement is *nil*.

Syntax:

RepeatStat = **repeat** SS **until** SS **end**

Example:

```
repeat x = x * 2 until x > k end
```

Try Statement

Execution of an expression may terminate in a failure or exception. We say that the expression has *thrown* an exception. Exceptions are implemented with objects in WebL. The *throw* function accepts any object to throw as an exception. The try expression is used to trap a failed expression, or more commonly said, to catch the exception object. In case no exception occurs, the try statement simply executes a statement sequence and returns its value. In the case of an exception occurring in the statement sequence, a sequence of guarded expressions is evaluated. The guards are evaluated in sequence until one evaluates to true, whereafter the associated expression is evaluated and returned as value. In case no guard evaluates to true, the exception is automatically re-thrown (and may be caught by an enclosing try state-

ment. The exception object is automatically assigned to a programmer-specified exception variable. (WebL will automatically declare the exception variable in a fresh context.)

By definition any WebL object can be thrown as an exception. By convention though, most exception objects consist of a string-valued field *msg* that describes the exception, and a string valued field *type* that is used for identifying the exception type. In some cases, the exception object contains fields that give more specific information on what went wrong, for example the file and line number where it occurred.

Table 47 on page 183 lists the exceptions thrown by statements, operators and built-in functions of the WebL language.

Syntax:

CatchStat = **try** SS **catch** Ident { **on** E **do** SS } **end**

Examples:

```
try
   p = GetURL("http://www.yahoo.com")
catch E
   on E.statuscode == 404 do
      PrintLn("page not found")
   on E.type == "HttpError" do
      PrintLn("connection error")
   on true do
      nil              // catch everything else
end

Throw ([. type="OutOfMemory", msg="No space left" .])
```

Every Statement

The *every* statement enumerates the elements of sets, lists, strings, objects and piece-sets. (Piece-sets will be introduced later.) Set elements are enumerated in an undefined sequence. List elements are enumerated from left to right. Enumerating a string gives the individual characters of the string from left to right. Enumerating an object gives the field names of the objects.

While enumerating, each element is assigned in turn to the loop variable and the body of the every statement executed. The value of the every statement is *nil*.

Syntax:

EveryStat = **every** Ident **in** E **do** SS **end**

Example:

```
every x in [1, 2, 3, 4] do
    PrintLn("X has the value ", x)
end
```

Lock Statement

The *lock* statement is used to prevent race conditions in muli-threaded programs. The statement *locks* an object, executes a statement sequence, and *unlocks* the object. The lock on the object can only be held by a single thread at any specific time. In case a thread tries to aquire a lock on an object held by another thread, the thread is suspended until the lock is released.

Syntax:

LockStat = **lock** SS **do** SS **end**

Example:

```
var counter = [.
    val = 0,
    inc = meth(s, i)
        lock s do
            s.val = s.val + 1
        end
    end
.]
```

Begin Statement

The *begin* statement allows the programmer to introduce a new statement sequence in a sub-expression. This is sometimes useful to open a fresh context in which temporary variables can be declared.

Syntax:

BeginStat = **begin** SS **end**

Example:

```
x + begin var s = a * b; s * s end
```

Return Statement

The *return* statement returns the value of a function or method call. The *return* token is optionally followed by an expression that evaluates to the value returned (otherwise *nil* is returned by default). Note that the return statement can be used to return a value *early*. Contrast the WebL convention of the last expression of a function or method calculating the return value, which allows returning a value only at the end of the function or method. Also note that it is a runtime exception to execute a return statement outside of a function or method body.

Syntax:

ReturnStat = **return** [E]

Example:

```
var F = fun(s)
    if s == nil then
        return ""
    end;
    ToString(s)
end;
```

Built-in Functions

Several functions are built into the WebL programming language (in contrast to functions written by the programmer). We distinguish between *normal built-ins* and *special built-ins*. Normal built-ins evaluate all their actual arguments before invoking the function. Special built-ins defer the evaluation of their arguments to the function being invoked. Examples of special built-ins include *Time*, *Timeout*, and *Retry*.

Most built-ins accept only a fixed number of arguments. Some built-ins like *PrintLn* accept any number of arguments of any value type. Variable length argument builtins are specified with ellipses (...) in Table 13. An actual argument can be of any value type if no explicit type is given in the table. The pseudotype *any* denotes values of any type.

As a shorthand we sometimes use the notation

```
argname: {type1, type2, ... }
```

to indicate that *argname* can be of *type1* or *type2*. (See Table 15 on page 64.)

TABLE 13. Core Built-in Functions

Function	Description
Assert(x: bool)	Throws an assertion failed exception if *x* is false.
Boolp(x): bool Charp(x): bool Funp(x): bool Intp(x): bool Listp(x): bool Methp(x): bool Objectp(x): bool Realp(x): bool Setp(x): bool Stringp(x): bool Pagep(x): bool Piecep(x): bool Tagp(x): bool Piecesetp(x): bool	Predicates that check if a value is of a specific type.
Call(cmd: string): string	Executes a shell command and returns the output written to standard out while the command is running. The command string may contain references to variables in lexical scope by writing *$var* or *${var}*. The value of these referenced variables are expanded before the command is executed.
Clone(o: object, p: object, ...): object	Makes a new object by copying all the fields of the objects passed as arguments. Fields of *p* have precedence over fields of *o* (and so on). The field ordering of the resulting object is defined by enumerating the fields of *o*, *p*, and so on in that sequence.
Error(x, y, z, ...): nil	Prints arguments to standard error output.
ErrorLn(x, y, z, ...): nil	Prints arguments to standard error output followed by end-of-line.
Eval(s: string): any	Evaluates the WebL program coded in string *s*.

TABLE 13. Core Built-in Functions

Function	Description
Exec(cmd: string): int	Executes a shell command and returns the exit code returned by the command. The command string may contain references to variables in lexical scope by writing *$var* or *${var}*. The value of these referenced variables are expanded before the command is executed.
Exit(errorcode: int)	Terminates the program with an error code.
DeleteField(o: object, fld): nil	Removes the field *fld* from the object *o*.
First(l: list): any	Returns the first element in a list.
GC(): nil	Explicitly invokes the Java garbage collector.
Native(classname: string): fun	Loads a WebL function[a] implemented in Java.
Print(x, y, z, ...): nil	Prints arguments to standard output.
PrintLn(x, y, z, ...): nil	Prints arguments to standard output followed by end-of-line.
ReadLn(): string	Reads a line from standard input (throws away the end-of-line character).
Rest(l: list): list	Returns a list of all list elements except the first element.
Retry(x): any	Executes expression *x* and returns its value. In case *x* throws an exception, *x* is re-executed as many times as needed until it is successful.
Select(l: list, from: int, to: int): list	Extracts a sublist of l starting at element number *from* and ending at element number *to* (exclusive).
Select(s: string, from: int, to: int): string	Extracts a substring of starting at character number *from* and ending at character number *to* (exclusive).

TABLE 13. Core Built-in Functions

Function	Description
Select(s: set, f: fun): set Select(l: list, f: fun): list Select(p: pieceset: f: fun): pieceset	Maps sets, lists, and piecesets to sets, lists, and piecesets respectively according to a membership function *f*. Function *f* must have a single argument and must return a boolean value indicating whether the actual argument is to be included in the set, list or pieceset.
Sign(x: int): int Sign(x: real): int	Returns -1, 0, +1 if x < 0, x = 0, and x > 0 respectively.
Size(l: list): int	Returns the number of elements in a list.
Size(s: set): int	Returns the number of elements in a set.
Size(s: string): int	Returns the number of characters in a string.
Sleep(ms: int): nil	Suspends thread execution for the specified number of milliseconds.
Sort(l: list, f: fun): list	Sorts the elements of *l* according to the comparison function *f*. The function f needs to take two formal arguments and return -1, 0, or +1 if the actual arguments are less, equal, or more than each other.
Stall()	Program goes to sleep forever.
Throw(o: object)	Generates an exception.
Time(x): int	Returns the time (in milliseconds) it takes to evaluate the expression *x*.
Timeout(ms: int, x): any	Performs the expression x and returns its value. If the evaluation takes more than the specified amount of time (in milliseconds), an exception is thrown instead.
ToChar(c: char): char	No operation.
ToChar(i: int): char	Converts an integer to the equivalent Unicode character.

TABLE 13. Core Built-in Functions

Function	Description
ToInt(c: char): int	Returns the Unicode character number of a char.
ToInt(i: int): int	No operation.
ToInt(r: real): int	Truncates the real value to an integer (rounding towards zero).
ToList(s: set): list ToList(l: list): list ToList(s: string): list ToList(o: object): list ToList(p: pieceset): list	Enumerates all the elements of the argument and returns a list. (See "Every Statement" on page 38.)
ToInt(s: string): int	Converts a string to the numeric equivalent.
ToReal(c: char): real	Same as *ToReal(ToInt(c))*.
ToReal(i: int): real	Converts an integer to a real.
ToReal(r: real): real	No operation.
ToReal(s: string): real	Converts a string to a real value.
ToSet(s: set): set ToSet(l: list): set ToSet(s: string): set ToSet(o: object): set ToSet(p: pieceset): set	Enumerates all the elements of the argument and returns a set. (See "Every Statement" on page 38.)
ToString(x): string	Converts a value to its string representation.
Trap(x):object	Executes *x* and returns the exception object that was caught. In case no exception is thrown in *x*, *nil* is returned. In addition, the exception object contains a field *trace* that has extra information why the exception occurred. This information is useful for logging unexpected exception events in your WebL programs.
Type(x): string	Returns the type of x (nil, int, real, bool, char, string, meth, fun, set, list, object, page, piece, pieceset, tag).

a. The class indicated must be a subclass of *webl.lang.expr.AbstractFunExpr*

Modules

To facilitate the reuse of code, WebL allows you to package commonly used routines in a *module*. An example module might be routines to process pages from a specific web server. *Client* programs can access the routines by *importing* the module. This is indicated by the client writing an *import* statement specifying all the modules that a program requires. After importing a module, the variables declared in that module can be accessed. This is done by writing the module name, followed by an underscore character and the variable name. For example, the following program imports module A, accesses a variable and calls a function in that module:

```
import
   A;

PrintLn("The value of x in A is ", A_x);
A_Doit()
```

The import statement may occur only at the beginning of a program. Imported variable references must always be explicitly qualified by the module name and an underscore. Note the choice of the underscore character allows us to separate the variable name space and module name space (i.e. a module and a variable might have the same name).

One of the side-effects of importing a module is the loading of the module into memory. WebL keeps track of all loaded modules in a global *module list*. Before importing a module, the module list is checked to see whether the module has been loaded before — if so, the previously loaded module is reused. Thus a module can be loaded only once. There is no operation to *unload* a module.

A module is nothing more than a statement sequence stored in a file with the extension ".webl". *Loading* a module involves executing this statement sequence (once). The language allows the programmer to *export* declared variables from the module. The exported variables are visible to clients of the module. Unexported variables cannot be accessed from clients. Exported variables are only allowed in the top level context. For example, in the following implementation of module *A* (which must be stored in the file "A.webl") the variable *y* is hidden from clients:

```
// Implementation of module A
export var x = 42;
var y = 10;

export var Doit = fun() PrintLn("ok.") end
```

Modules may import other modules. This allows the construction of a *module hierarchy* in the form of an *import graph*. Note that the graph is directed and a-cyclic — recursive module imports are not allowed and will cause a runtime error.

Service Combinators

We can imagine that many things can go wrong with a computation on such a large distributed scale as the World Wide Web. For example, part of a WebL computation might fail because of a failed web server or missing web page. Thus the unpredictable nature of the web causes many more exceptions than in a non-distributed environment. To counteract this problem, WebL provides a few convenient ways to handle exceptions. The exception handling mechanism is based on a formalism called *service combinators*. In this formalism we talk about *services* — computations that depend on remote web servers — that complete *successfully* or *fail* (throw an exception).

The service combinators allow several services to be combined in ways that can make a computation more reliable and in some cases even improve its speed. Note that by service we mean *any* WebL computation. WebL supports several service combinators: *sequential execution*, *concurrent execution*, *time-out*, *repetition* and *non-termination*.

One of the most basic services involves fetching a page from the Web. To make the examples that follow more realistic, we are going to use two of these built-in functions. More details about the exact behavior of these functions can be found in “Retrieving Page Objects” on page 59. Note that any WebL computation can be regarded as a service.

Services

```
GetURL(url, [. param1=val1, param2=val2, ... .])
PostURL(url, [. param1=val1, param2=val2, ... .])
```

The *GetURL* function fetches with the HTTP GET protocol the resource associated with the URL. It returns a page object that encapsulates the resource. The function fails if the fetch fails. The second argument to *GetURL* provides the server with

query arguments. A similar function called *PostURL* uses the HTTP POST protocol, used to fill in Web-based input forms.

```
// This program simply attempts to fetch the named URL.
page = GetURL("http://www.digital.com")

// This program looks up the word "java" on the
// AltaVista search engine.
page = GetURL(
    "http://www.altavista.digital.com/cgi-bin/query",
    [. pg="q", what="web", q="java" .])
```

Sequential execution *S ? T*

The "?" combinator allows a secondary service to be consulted in case the primary service fails for some reason. Thus, the service *S ? T* acts like the service *S* except that if *S* fails then it executes the service *T*.

```
// This program first attempts to connect to AltaVista
// in California, and in the case of failure,
// attempts to connect to a mirror in Australia.
page = GetURL("http://www.altavista.digital.com") ?
    GetURL("http://www.altavista.yellowpages.com.au")
```

Concurrent execution *S | T*

The "|" combinator allows two services to be executed concurrently. The service *S | T* starts both services *S* and *T* at the same time and returns the result of whichever succeeds first. If both *S* and *T* fail, then the combined service also fails. Should one service complete before the other, the slower service is stopped. Stopping the slower service is performed in a controlled manner, to ensure the run-time remains in a consistent state. Typical checkpoints at which WebL will stop a service is at function or method call boundaries, and at the beginning or end of programmed loops.

```
// This program attempts to fetch a page from one
// of the two alternate sites. Both sites are
// attempted concurrently, and the
// result is that from whichever site
// successfully completes first.
page = GetURL("http://www.altavista.digital.com") |
    GetURL("http://www.altavista.yellowpages.com.au")
```

Time-out *timeout(t, S)*

The time-out combinator allows a time limit to be placed on a service. The service *Timeout(t, S)* acts like *S* except that it fails after *t* milliseconds if *S* has not completed within that time. *S* will be stopped in controlled manner when it times-out (see the concurrent execution description above for details on how services are stopped).

```
// This program attempts to connect to
// alternative AltaVista mirror sites,
// but gives a limit of 10 seconds to succeed.
page = Timeout(10000,
    GetURL("http://www.altavista.digital.com") |
    GetURL("http://www.altavista.yellowpages.com.au "))
```

Repetition *Retry(S)*

The repetition combinator provides a way to repeatedly invoke a service until it succeeds. The service *Retry(S)* acts like *S*, except that if *S* fails then *S* starts again. The loop can be terminated by writing *Timeout(t, Retry(S))*.

```
// This program makes repeated attempts in the
// case of failure, alternating between two services.
page = Retry(
    GetURL("http://www.x.com") ?
    GetURL("http://www.y.com"))
```

Non-termination *Stall()*

The stall combinator never completes or fails.

```
// This program repeatedly tries to fetch the URL, but
// waits 10 seconds between attempts.
page = Retry(getpage("http://www.digital.com") ?
    Timeout(10000, Stall())
```

CHAPTER 3

Pages

To set the stage for the next chapter on markup algebra, we must introduce fetching a page from the Web and mapping the page into structures compatible with the WebL language. Although we cannot give a thorough overview of the Web protocols and formats involved in this process, we present a short tutorial to introduce the particular vocabulary used in WebL. Thus the most of this chapter is a review of things that might be known to many readers. However, the chapter does contain important information and definitions that will be required in the following chapters.

Basic Protocol Terminology

The World Wide Web (WWW) consists of a large number of Web sites, domains or servers that provide services to clients over the Internet. The typical clients of these services are users who retrieve web pages with a software application called a Web browser. In contrast, WebL is a client that fetches pages in an automated manner under the control of a program. The purpose of this section is to show how this works.

Uniform resource locators. Pages are identified by a uniform resource locator or URL. A URL identifies the web site, the location of the page on the web site, the

filename of the page, and the Internet transmission protocol required to fetch the page. Much simplified, URLs have the following form:

```
http://hostname/path/filename.html
```

Here *http* refers to the protocol being used, *hostname* the web site or machine identification, and *path* the directory on that machine where the page called *filename.html* is stored.

HTTP. The Hypertext Transfer Protocol transfers the page over the Internet. The basic steps are:

- Establish a communications link from client to the web server identified by host name.
- The client sends an HTTP *request* to the server. The request consists of the location or filename of the page to retrieve (/path/filename.html), *headers*, and optional *parameters.*
- The server answers with an HTTP response. The response consists of a status code (indicating success or failure), a *status* message, *headers*, and the page *data* itself (contents of */path/filename.html* on the server).
- The connection is closed.

Request parameters provide additional information to a web server about the requested data. This information is often used to access a special service on the server that generates appropriate responses dynamically, for example by looking up data in a database. Each parameter consists of a parameter *name* and a *value*. Parameters are included in the HTTP request in one of two methods:

- The HTTP GET request appends the parameters (encoded in a special way) to the URL,
- The HTTP POST request appends the parameters to the end of the request.

GET requests issued with parameters are recognized by a question mark (?) followed by the parameters (name value pairs) appended to the URL. In contrast, parameters of a POST request are hidden and not visible from the URL.

POST requests are the preferred method for transmitting the contents of an HTML fill-in form to a web server. Their main advantage is that larger amounts of data can be submitted than with the GET method. Note however that the GET method is also applicable to fill-in forms, and is typically used when parameters are few and relatively short. The GET method is also the default when no parameters are passed.

WebL supports both the GET and POST methods as built-in functions. These functions accept the request parameters in a WebL object and perform the correct encoding and packing in the HTTP request (either in the URL or at the end of the request).

Parameter encoding. For each parameter with name N and value V, we construct a string "N=V". All parameter strings are then concatenated (separated by a & symbols), and a question mark is prepended. The URL of a GET request with parameters will thus have the general form:

```
http://domainname/path/filename.hml?N1=V1&N2=V2&N3=V3
```

Names consist of alpha-numeric characters. Values may contain any character except those that are reserved for URLs. To encode the latter characters, we replace them with a percentage sign (%) followed by a two-digit hexadecimal number specifying the ASCII code of the character. In addition, spaces are replaced by *plus* (+) signs.

Request and response headers. HTTP *request headers* give the web server more information about the request itself, the browser that is being used, etc. HTTP *response headers* give the browser more information about the page that is returned. In contrast to parameters that can be freely picked, headers are predefined by the HTTP protocol. A header consists of a *name* and a *value*. Although WebL can add request headers and read response headers, scripts seldom need to exercise this control. The main uses of this feature include mimicking a specific web browser model, and retrieving and setting cookies.

MIME types. One of the important pieces of information returned by an HTTP response is the type of the data that is being retrieved (included in a response header). The MIME type specifies if the data is an HTML page, an XML page, an image, a Postscript file etc. WebL supports onlythe MIME types corresponding to what it can parse: Plain text, HTML and XML. Attempting to process anything else in WebL causes an exception. A common MIME type is the one that identifies HTML documents, typically written in one of the following forms:

```
text/html
text/html; charset="us-ascii"
text/html; charset=ISO-8859-1
```

The *charset* parameter is optional — it indicates the character encoding (or content encoding) the document is encoded in. WebL uses the *charset* parameter (or makes

an educated guess as to its value when missing) to determine how pages are converted into an internal Unicode format.

Unfortunately, many web servers do not return the correct MIME type information for certain documents, which makes it impossible for WebL to parse the document. To prevent this from occuring, it is possible to *override* the MIME type of a document explicitly when using the *GetURL* and *PostURL* builtin functions.

Cookies. Many web servers today use “cookies” to store client-side state. For example, a typical application of cookies is to unique identify customers at a web store front. At startup time WebL knows about no cookies (i.e. the cookie database is empty). As cookies are set by servers during HTTP requests, the cookie database will fill up. Each WebL HTTP request is checked against the cookie database, and if necessary, WebL will return the appropriate cookies to the server. A special WebL module called *Cookies* allow you to save the cookie database to a file, and reload it at a later time.

Page parsing. Once an HTTP request is completed, WebL parses the page data into an internal format that makes it easy to query and manipulate the page.

WebL programmers should have high-level understanding of how HTML and XML are handled — to this end the following section gives an overview of basic markup concepts and how they relate to WebL. This background material is a prerequisite for the following chapter on the search algebra and page manipulation features.

Markup

The *Hypertext Markup Language* (HTML) and *Extensible Markup Language* (XML) are both instances of the *Standard Generalized Markup Language* (SGML). SGML was conceived in the middle 1980's as a text markup notation for exchanging hierarchically organized electronic documents. SGML consists of two parts, namely the document markup in the form of *tags*, and a meta-description of a document class called a *Document Type Definition* (DTD).

DTD's are typically designed for special purposes, and define the tag names, tag structures, and hierarchical organization of documents that conform to the DTD.

HTML, as an instance of SGML, consists of a DTD that defines the exact version of HTML being used, and a set of conventions followed by web browsers for rendering the markup on a computer display. Most of the HTML involves how markup should be presented, for example what fonts are used and in what size, colors, spacing, line breaks, and so on. The main clients of HTML are real people viewing the pages marked up in this manner.

In contrast, XML is an instance of SGML for the exchange of *content* or *application specific data* over the web. The idea is that if two or more people can agree on a common DTD (that is the markup and structure of a document), they can exchange documents and other information. In a simplistic way XML can be regarded as a variant of HTML where you may define your own markup. The main clients of XML are programs that process the content of web pages — although XML can be viewed in a browser, it has nothing to do with *presentation.*

XML documents are typically grouped according to the DTD that is used. For example, XML documents using the *Content Definition Format* (CDF) DTD are used in push media, and XML documents using the *Chemical Markup Language* (CML) DTD are used to exchange molecular structures.

Tags. At the simplest level, pages consist of sequences of characters (called *parsed character data* or *PCDATA*) and markup symbols called *tags.* Tags consist of characters enclosed between less than (<) and greater than (>) symbols. The tag contents specify a tag name and optionally any number of *attributes*. Tag names are predefined in HTML (for example, *H1*, *H2*, *P*, *FONT*, *TITLE*, etc) whereas XML tags are defined according to a DTD. Attributes consist of (name, value) pairs.

The general tag style is as follows:

```
<name A="abc" B="123">
```

Here the tag name name is followed by attribute values A and B, having the values *abc* and *123* respectively. Values are always quoted by single or double quotes in XML; HTML values have a more flexible syntax allowing certain values to be unquoted. HTML also allows attributes to have no value, for example:

```
<name A B>
```

Elements. A hierarchical structure is imposed on a page by collecting tags and parsed character data into *elements*. We distinguish between comment elements, (non-empty) elements, empty elements, processing instruction elements, and SGML directive elements.

Note that in this manual we diverge from SGML terminology so as to explain the unified view WebL presents to the programmer by mapping names and attributes of elements to *piece* objects (introduced later) and object *fields*. The *name* of a piece is derived from the start tag of the element.

Comments. Comment elements specify text to be ignored during document parsing. Comments consist of a single tag in the following style:

```
<!-- this is the comment text -->
```

The *name* of a comment element is "!--". In WebL, the comment element has a field called *comment*, which has as value the text occuring between the “--” tokens.

Non-empty Elements. These consist of a start tag, any number of nested elements or PCDATAs, and a matching end tag. Everything between the start tag and end tag is said to be inside or contained in the element. The general format is as follows:

```
<tagname A="abc" B="123"> ... </tagname>
```

The names of the start tag and end tag must match. Note how the end tag starts with a forward slash character. Only the start tag may have attributes. The name and attributes of non-empty elements are those of the start tag.

In the case of attributes with no value, the attribute of the element is set to the empty string.

Empty elements. Empty elements do not have any content, and thus do not require an end tag. They have the format:

```
<tagname A="abc" B="123" />
```

Note the forward slash that ends the tag. The element name and attributes are those of the tag.

Empty elements appear only in XML documents. HTML has something similar to an empty element but it cannot be distinguished from a start tag. For example, the

HTML markup <*br*> does not have a corresponding end tag, and thus is equivalent to an empty tag. WebL knows about these anomalies by virtue of the HTML DTD.

Processing Instructions. Processing instruction elements give instructions to the page parser to perform special handling of its contents. They are used only in XML, and consist of a single tag:

```
<?tagname ... ?>
```

Here the ellipses take the place of the processing instructions. Similar to comment elements, "?tagname" is defined as the name of the element. The processing instructions (between the question marks) are mapped onto the *content* field of the element).

SGML Directives. SGML directives provide information about the DTD of the page being parsed. They have the form:

```
<!tagname ... >
```

Here the ellipses take the place of the directive. As before, "!tagname" is defined as the name of the element, and the element has an attribute called *content* that stores the directive. The most commonly occurring SGML directive is *!DOCTYPE,* which specifies the name of the DTD to be used to parse the remainder of the document.

Optional tags. Parsing of HTML is made complicated by an SGML feature called optional tags (a feature that has explicitly been left out of XML). The idea is that the DTD often gives enough contextual information to infer that a start or end tag must be present at a certain position in the document. For several HTML elements, either start or end tags are declared to be optional, and should be inserted automatically by the parser. For example, the paragraph <P> element is in fact a non-empty element that has a corresponding </P> end tag. However, most HTML documents do not contain these optional end tags, in which case the HTML parser has to infer where paragraphs end. WebL knows the HTML DTD, and can thus insert optional tags when needed. In general, WebL attempts to make a faithful internal representation of the documents it parses (including spaces and new lines), except for the fact that it inserts optional tags when appropriate. Conversion from the internal format to external format might thus result in slightly different (but equivalent) pages.

Character Entities. Inserted in the PCDATA stream we often find character entities of the form *&...;* (where ... stands for a number or an alphanumeric name) denoting special symbols. For example, *<* and *>* denotes the less than and

greater than symbols. This encoding is used both to embed special symbols that might be confused with markup, and to provide a human-readable way to represent all Unicode characters. WebL does not perform any translation of character entities by default when fetching a page, but does provide a built-in called *ExpandCharEntities* to process them afterwards, and an retrieval option that switches on expansion (Table 16 on page 65).

Case-sensitivity. XML is case-sensitive and HTML is case-insensitive. In the case of XML, WebL keeps all element names and attributes in their original case. In the case of HTML, WebL converts all element names and attribute names to *lowercase*.

URL resolution. Many elements have attributes that specify URLs of other documents on the Web. Most of these URLs are specified relative to the document itself. WebL simplifies handling of these URL attributes by *resolving* them to an absolute URL when the document is fetched. To determine which attributes refer to URLs, WebL uses slightly modified HTML DTDs internally that explicitly denote which attributes of elements contain URLs. No URL resolution is performed for XML documents. HTML URL resolution can be switched off with a page retrieval option (Table 16 on page 65).

Bad HTML. A surprisingly large number of pages on the web contain errors. Some of the typical errors encountered include:

- Forgotten end or start tags,
- Illegal nesting of elements forbidden by the DTD,
- Non-hierarchical markup where elements overlap instead of nest,
- The DTD specified by the DOCTYPE SGML directive does not match the markup the document contains,
- Tags with illegal names, etc.

WebL tries to take all these problems into account. In SGML terms, WebL is a non-validating processor. WebL only uses DTDs to correct simple mistakes and to add optional tags where needed. WebL also corrects overlapping tags in HTML to ensure that we have a hierarchically structured document. In general, we try to make as few changes as possible to the page with the guidance of the DTD. It is thus important to realize that when bad HTML or XML is parsed, the internal representation might not be what you expect from viewing the page source. To give users an idea of what WebL sees, a pretty printing function called *Pretty* is included in WebL that displays a representation of the parsed web page in a nicely formatted

way. We recommend using this tool as it often illustrates the badly formatted markup found on the web.

Badly encoded scripts. A growing number of web pages contain code written in scripting languages like JavaScript and VBScript. WebL attempts to skip over the contents of these parts of a page, so that the HTML parser does not get confused when seeing things that look like markup that are encoded in scripts. Typically this is not a problem, since authors are expected to always place scripts inside HTML comments <*!*-- ... > between the tags <*script*> and <*/script*>. Unfortunately this advice is sometimes ignored, and the script code is left uncommented, which can confuse the page parser. For this reason, WebL does not parse enything at all between <*script*> and <*/script*> tags in HTML pages. The whole stretch of text between the two tags remains a single unparsed text segment.

Retrieving Page Objects

WebL's internal representation of a web resource is a *page object*. The built-in functions *GetURL* and *PostURL* fetch a page from the Web, and return a page object. In the next chapter we will introduce functions that will turn a page object into a string value and back, search and manipulate markup in interesting ways, etc.

The *GetURL* and *PostURL* functions take a variable number of arguments that specify the URL to be fetched, request parameters, additional headers, and options. (See Table 15 on page 64.)

WebL's ability to process different URL protocols like *http*, *file*, and *ftp* is inherited from the underlying Java implementation (i.e. WebL does not provide support for any additional protocols). The most common URL used in WebL is the one corresponding to the HTTP protocol. Note that page redirects and cookies are handled transparently by WebL, but this default behavior can be overridden if required.

The request parameters passed to the functions are in the form of objects. For example, the URL of a typical AltaVista request has the following form:

```
http://www.altavista.digital.com/cgi-bin/
query?pg=q&what=web&kl=XX&q=%22Hannes+Marais%22
```

This can be converted into a call to the *GetURL* function in the following manner:

```
GetURL(
    "http://www.altavista.digital.com/cgi-bin/query",
    [. pq="q", what="web", kl="XX",
    q="\"Hannes Marais\"" .])
```

Parameters. WebL will automatically take care of packing request parameters in the correct way as required by the GET and POST protocol variants. In the case of a *PostURL* request, the correct construction of the parameter object needs to be deduced by the programmer from the Web form where the request originates from. (This is beyond the scope of this manual.)

Note that the HTTP specification requires that the POST parameters be submitted in the order they appear in the form on the page. It is thus important to list the *param* object fields in the same order as the fields in the form (recall that object fields are ordered according to definition sequence).

There are two tricks that are sometimes needed when submitting form data. The first trick involves posting multiple parameters that have the same name. An HTML form might allow the user to pick several options from a list (for example to indicate his or her favorite programming language). This can be specified as follows:

```
PostURL("http://...",
    [. gender="male",
    language=["WebL", "Java"] .])
```

Note the use of a field of type list to indicate the multiple values. For those readers familiar with the HTTP specification, the data that will be posted as follows in the body of the HTTP request:

```
gender=male&language=WebL&language=Java
```

Note how the parameter language appears twice in the submitted data.

The second trick is a work-around for the case when the submitted parameters do not match well with the WebL object type. This might for example be the case when the server does not conform to the HTTP specification, and allows the posting of data in any format. To handle this case, you may pass a string type instead of an object type to the *param* argument of *PostURL*. Of course, in such a case you have to take care yourself of encoding the parameters correctly (and for this "Module Url" on page 138 is useful). Keeping with our example above, this would be coded as:

```
PostURL("http://...",
   "gender=male&language=WebL&language=Java")
```

Also note that the *GetURL*, *Files_GetURL*, and *Files_PostURL* functions also accept a string argument instead of an object for parameters (Also see “Module Files” on page 121). In the case of the *Get* variants of the functions, the parameter string is simply appended to the URL itself (with WebL adding the usual "?" in between).

Headers. The *GetURL* and *PostURL* functions add extra HTTP header fields to a request in case the optional *header* object is used as an argument. Headers that might need to be added in this way can be the client identification, cookies, etc. The response headers of a request (with names converted to lowercase) become part of the page object returned by the functions. For example, the program:

```
var p = GetURL("http://www.digital.com");
PrintLn(p);
```

prints the page fields:

```
[. "server" = "Apache/1.2.4",
"connection" = "close",
"date" = "Fri, 01 May 1998 19:45:47 GMT",
"content-type" = "text/html",
"URL" = "http://www.digital.com/" .]
```

Some HTTP response headers, like for example *Set-Cookie*, might be repeated several times. In such a case, the value of the header field will be list of string values, in the order of occurrence in the HTTP response. The page fields of an HTTP response with multiple headers of the same name might thus look as follows:

```
[.
"content-type" = "text/html",
"URL" = "http://www.digital.com/",
"Set-Cookie" = ["id=123", "pw=abc" ]
.]
```

The same idea is applied when submitting multiple headers with the same name in a request[1]:

1. Although WebL supports multiple request headers, the underlying Java implementation does not.

```
GetURL(url, nil, [.
                    HeaderA ="xyz",
                    HeaderB = ["id=123", "pw=abc"]
                    .] );
```

Overrides. Unfortunately, many servers return the incorrect MIME type for a page. This incorrect MIME type can be overridden by passing a *mimetype* field in the *options* object argument of *GetURL* and *PostURL*. The *mimetype* field of the options object must be of type string, and the value must be taken from Table 14. In case the content encoding is known, an optional *charset* MIME type parameter can be specified (See "MIME types" on page 53.). For example, we can write the following to override the MIME type of page *X* to be of type plain text:

```
GetURL(X, nil, nil, [. mimetype="text/plain" .])
```

TABLE 14. Supported MIME Types

MIME Type	Parser Used
text/plain	Plain text
text/html	HTML
text/xml	XML
application/xml	XML

A related problem that we often face when processing HTML pages, is that the author of the page either gives no indication which version of HTML is being used, or gives an incorrect indication. The HTML version information is part of the DOCTYPE tag, and identifies the HTML DTD to be used to parse the page. WebL relies on this information to parse an HTML correctly. In case of an incorrectly authored page, the DTD can be explicitly overridden by the WebL programmer by adding a *dtd* field to the *options* object argument. The value of the parameter should be the officially assigned named of the DTD. For example, the following option values identify HTML 4.0, 3.2, and 2.0:

```
[. dtd="-//W3C//DTD HTML 4.0//EN" .]
[. dtd="-//W3C//DTD HTML 3.2//EN" .]
[. dtd="-//IETF//DTD HTML//EN" .]
```

The fields of the *option* argument to *GetURL* and *PostURL* are summarized in Table 16 on page 65.

TABLE 15. Functions to Retrieve Web Pages

Function	Description
GetURL(url: string): page	Uses the HTTP GET protocol to fetch the resource identified by the URL.
GetURL(url: string, params: {object,string}): page	The *params* object/string contains the parameters of a GET that includes a query.
GetURL(url: string, params: {object,string}, headers: object): page	The *headers* object specifies the additional headers to include in the GET request.
GetURL(url: string, params: {object,string}, headers: object. options: object): page	The *options* object allows, amongst other functions, the overridng of the MIME type and DTD to be used for parsing the page.
PostURL(url: string): page	Uses the HTTP POST protocol to fetch the resource identified by the URL.
PostURL(url: string, params: {object,string}): page	The *params* object/string contains the parameters of a POST to fill in a web form.
PostURL(url: string, params: {object,string}, headers: object): page	The *headers* object specifies the additional headers to include in the POST request.
PostURL(url: string, params: {object,string}, headers: object. options: object): page	The *options* object allows, amongst other functions, the overridng of the MIME type and DTD to be used for parsing the page.
HeadURL(url: string): page	Uses the HTTP HEAD protocol to fetch the resource headers identified by the URL.
HeadURL(url: string, params: {object,string}): page	The *params* object contains the parameters of the HEAD request.
HeadURL(url: string, params: {object,string}, headers: object): page	The headers object specifies the additional headers to include in the HEAD request.

TABLE 16. Fields of the option object

Field	Description
autoredirect	Controls whether moved pages (for example HTTP status code 302) get automatically fetched from their new locations. The default value is *true*.
charset	Overrides the character set used to parse the document. Typical values are “ISO-8859-1”, “UTF8”, etc.
dtd	Overrides the DTD to be used when parsing the page. The value of this field must be string with the official DTD name as defined in the SGML catalog.
emptyparagaphs	When this flag is set to *true*, the HTML parser will regard paragraphs (i.e. tags) as empty markup elements instead of the usual ... pairs. (The is an example of another empty markup element). This option is sometimes useful when confronted with pages where is used without regard for the HTML specification (for example, the incorrect use of inside <font>, and so on). The default value of this flag is *false*.
expandentities	When this flag is set to *true*, character entities like """ etc are expanded as the page is parsed. The expansion is onlye performed on HTML pages, and between markup tags (i.e. not inside attributes). The default value of this flag is *false*.

TABLE 16. Fields of the option object

Field	Description
fixhtml	When this flag is set to true, the HTML parser attempts to correct incorrectly nested HTML elements in a page (for example, putting a H2 inside a H1). This has the effect of regularizing badly formatted HTML, at the cost of sometimes unintutive parses. The default value of this flag is *false*.
mimetype	Overrides the mime type to be used when parsing the page. See Table 14 on page 63 for typical string values this field may assume.
noncompliantPOSTredirect	When this flag is set to *true*, HTTP POST requests that are *redirected* by a web server to another URL, are automatically changed into a subsequent HTTP GET request to that URL (a behavior which non-compliant with section 9.3 of the HTTP 1.0 specification and section 10.3 of the HTTP 1.1 specification). Note that all POST request parameters are ignored for the subsequent HTTP request. The default value of this flag is *true*, as many web browsers do not follow the specifications correctly in this regard.
resolveurls	When this flag is set to *false*, the URLs in the page are not resolved to absolute form. The default is *true*.
cookiedb	A string value specifying which cookie database to use for the request. See “Mutiple Cookie Databases” on page 117.

CHAPTER 4

The Markup Algebra

The WebL *markup algebra* is used for manipulating web pages and extracting data from them. Extracting information may range from simple operations like iterating all the links in a page to more complex operations that fill in Web forms and process the results returned from a server. Manipulating Web pages might involve rewriting parts of a page (for example highlighting words) or creating a new page from parts of several other ones.

The markup algebra consists of several operators and functions that operate on pages, tags, pieces and piece sets. There are operators and functions to create or build piece sets from pages or from other piece sets, convert pieces to their string representation, modify the content of a page, and so on.

Pages, Tags, Pieces, and Piece Sets

After a page is retrieved from the Web and parsed according to its MIME type, the page and its content is accessible for further computation in WebL. The computation that can be performed on a page is determined by the WebL *markup algebra*.

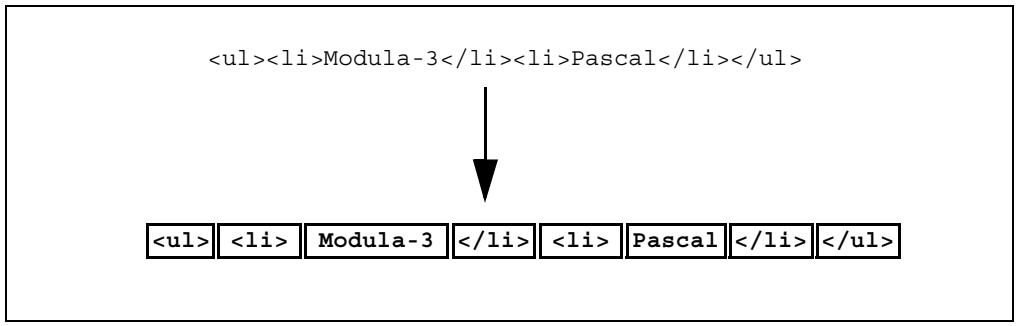
The markup algebra is based on three concepts: *tags*, *pieces* and *piece sets*. In simple terms, a *tag* corresponds to a markup tag, a *piece* identifies a contiguous subregion of a page, and a *piece set* is a collection of pieces.

Tags

The first step in parsing a Web page is to identify of all the markup tags in the page (enclosed between '<' and '>' characters). Each of the tags is converted into a *tag* (a WebL value of type *tag*). Conceptually the page then consists of a list of tag objects and text segments (or character data). We can use a simple train-like pictorial representation of a page to illustrate the conversion (Figure 1). In the figure, each box represents either a *tag* or a piece of *text*.

The WebL model also supports *unnamed* tags, the purpose of which will become clearer soon. The equivalent HTML or XML for an unnamed tag is '<>' (which of course does not occur in practice). WebL uses unnamed tags as place markers in a page. As their name suggests, unnamed tags do not have a name or attributes.

FIGURE 1. Converting Markup into Tag and PCData Sequences



Pieces

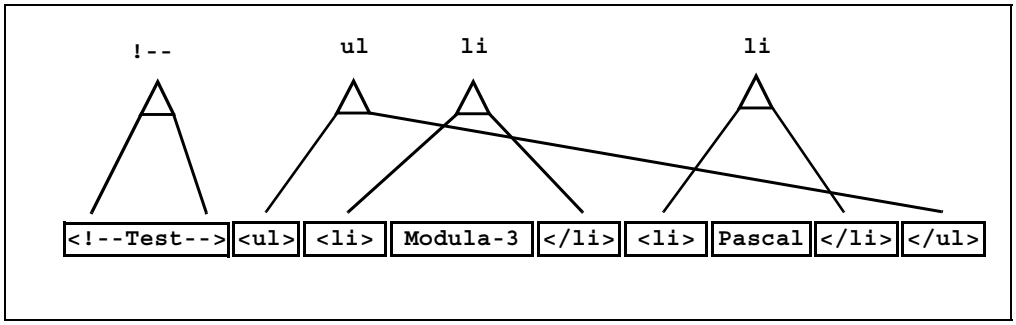
A *piece* is a WebL value type that denotes a region of page. Each piece refers to two tags: the *begin tag* that denotes the start of the region, and the *end tag* that denotes the end of the region. The region includes both the begin and end tag, and everything between them in the page. Note that the begin and end tag can also be the same. Another important fact is that pieces never point to text segments.

The most common types of pieces are those that correspond to elements in a page. We extend the box diagram notation to include triangles to denote pieces, and lines

from pieces to tags to denote begin and end tags (Figure 2). To allow the programmer to access element attributes, the attributes of an element's *begin tag* are copied into field variables of each piece. Thus, a piece is very similar to the object value type in that it looks and behaves in many ways like an object. Furthermore, we associate the appropriate *name* with each piece (in this case the names are the strings "!--", "ul", "li", and "li" written above the triangles).

Note how the begin and end tag of the comment piece refer to the same tag object. In accordance with our previous definition, a piece that refers to unnamed begin and end tags is called an *unnamed piece* (which correspondingly has the empty string as name).

FIGURE 2. Piece Notation



Piece Sets

As its name indicates, a *piece set* is a collection of pieces belonging to the same page. It is a set in the sense that a piece can belong only once to a piece set (but a piece can be a member of several piece sets). A piece set is also a list because pieces in a piece set are ordered. The piece ordering in a piece set is based on the begin and end tag positions of a piece in a page. We order pieces according to the left-to-right order of the begin tags.

Piece sets play a very important part in WebL. They form the basis of many of the operations that extract data from web pages.

Searching Functions

There are several ways in which piece sets can be created:

- Searching for markup elements by explicitly naming interesting elements (for example all the ‘a’ or ‘li’ elements).
- Searching for character patterns that match a regular expression.
- Searching for text segments.
- Searching for stylized sequences of markup patterns.
- Searching for segments delimited by explicitly named markup elements (i.e. paragraph extraction).

Element search

The *Elem* function returns a piece set of all elements that match a specific name. The function also allows the search scope to be restricted to a page or piece. Thus, a piece is constructed for each matching begin and end tag pair of a markup element with the indicated name in the indicated scope, and the resulting pieces are collected into a piece set result. For example, the following program fetches a page, calculates a piece set with all the *img* (image) elements of the page, and proceeds to print out the *src* attribute of each of those images:

```
var P = GetURL("http://www.nowhere.com");
var images = Elem(P, "img");
every image in images do
    PrintLn(image.src)
end
```

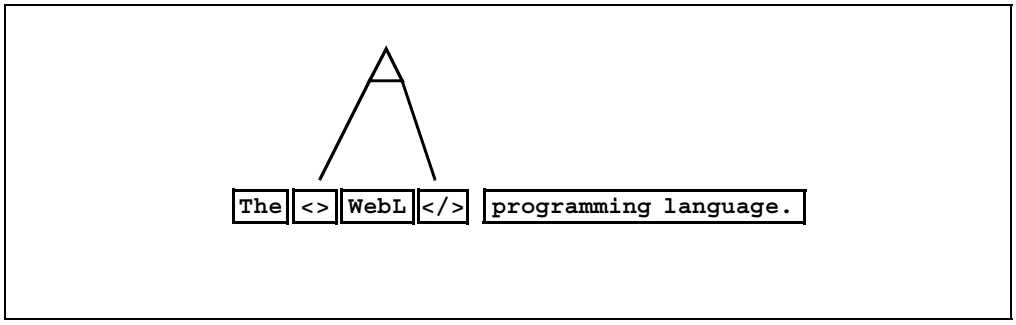
As can be seen from the example, the *every* statement also allows the iteration over the elements (the pieces) of a piece set.

Pattern search

The *Pat* function searches a page for character patterns that match a regular expression. The *Pat* function ignores the tag objects in a page — only the pure text stream is searched. For each occurrence of the pattern, a new piece is created. This involves inserting new unnamed tags just in front of and just after each pattern occurrence to keep track of the location. For example, Figure 3 shows how a page looks after searching for the word “WebL”. In this figure the unnamed tags are indicated by the boxes marked “<>” and “</>”.

The unnamed tags created while searching for character patterns are simply pattern locators — they are ignored by many operators and functions, and are automatically removed from the page when not required any more (in some sense they are “invisible”). Also, when a page is converted back to string format, the unnamed tags are removed. It is also important to know that unnamed tags are *always* inserted. Thus, searching for the same pattern twice will cause two nested and unnamed pieces to be inserted into the page. Another way of saying this is that tags are never *shared* by more than one piece.

FIGURE 3. Results of Searching for “WebL”



Pattern groups. The *Pat* function also supports Perl5 regular expression *groups*. Groups, as indicated with parenthesis in Perl5 regular expressions, identify constituent parts of the pattern to be matched. For example, a regular expression matching dates might have groups related to the day, month and year of the date. For each pattern matched in the page, the corresponding piece object of that pattern is attributed with fields numbered from 1 onwards that contain each of the groups occurring from left to right in the pattern. A field named 0 is also added to the piece which contains the complete matched pattern. For example, the following code fragment recognizes dates of the form “day-month-year”:

```
Pat(P, '(\d\d)-(\w*)-(\d*)')
```

The date pattern contains three groups, one for the two-digit day, one for a word representing the month, and one for the digits of the year. Given the occurrence of the string “20-Jan-1998” in a page, the corresponding piece object would look as follows:

```
[.
   0 = “20-Jan-1998”,
   1 = “20”,
   2 = “Jan”,
   3 = “1998”
.]
```

See page 182 for more details on the syntax of Perl5 regular expressions.

Once a piece set has been created with the *Elem* or *Pat* functions, we can apply WebL operators and functions to the result to perform further computation. For example, by indexing into a piece set with the indexing [] operator, we can extract the *nth* element of the piece set.

PCData search

The *PCData* function returns a piece set of all text segments that are contained in a page or piece. The name *PCData* is derived from the term “parsed character data”, which denotes the text segments on a page, i.e. what is left over when all markup tags are removed from a page. The *PCData* function is thus complementary to the *Elem* function, and somewhat related to the *Pat* function.

As an example, the following program fetches a page and prints out all the text segments occuring on the page (as delimited by markup tags):

```
var P = GetURL("http://www.nowhere.com");
every t in PCData(P) do
    PrintLn(Text(t))
end
```

(The *Text* function used above will be introduced a little later; it “prints” out the textual content of a piece.) Running this program will typically print a lot of white space; this is because the *PCData* function regards the empty regions between tags, for example, the area between *br* and *br* in the markup

```
some text<br>  <br>some text
```

as a distinct text segments. The following program shows how to get rid of these empty regions:

```
import Str;

var P = GetURL("http://www.nowhere.com");
every t in PCData(P) do
    var txt = Text(t);
    if Str_Trim(txt) != "" then
        PrintLn(txt)
    end
end
```

Note that the *PCData* function inserts new unnamed tags just in front of and just after each text segment to keep track of their location. This means that for the markup above, the piece identifying the empty region consists of an unnamed begin tag just *after* the first *br*, and an unnamed end tag just *before* the the second *br*.

Sequence search

HTML generated on-the-fly by web servers often contains highly stylized markup patterns without hierarchical structure. The markup might be a linear *sequence* of elements following each other. For example, we might expect an *H1* element, followed by a sequence of characters, followed by a *BR* element. We will be using this as our example in the following discussion.

Given a page and a string describing such a sequence (called a *sequence pattern*), the *Seq* function will return a piece set with all the occurrences of the sequence in the page. That is, each piece refers to an unnamed tag just before and after the first and last element of the sequence.

A sequence pattern is a list of element names separated by space characters. The intention is to match exactly that sequence of elements on the same element nesting level. It is important to note that sequence patterns do not match nested elements. For example, in our example, whether the *H1* element contains other elements is irrelevant.

To match sequences of characters, we use the # symbol. The # symbol will match the longest sequence of characters or unnamed tags at that position in the page. (Unnamed tags are ignored.)

The following will match all the *H1*, *text*, *BR* sequences in our example:

```
Seq(P, "h1 # br")
```

The *H1*, text, and *BR* pieces matched in each of the sequences are accessible by indexing the returned pieces (one for each sequence in the page) with integers from 0 onwards. For example, the following code fragment prints details of the matched sequences:

```
var S = Seq(P, "h1 # br");
every p in S do
    PrintLn("Heading=", p[0]);
    PrintLn("Text=", p[1])
end
```

Paragraph search

Paragraph search is one of the more complicated WebL page searching techniques; it is rather seldom used, but still performs a useful function that is sometimes required. The purpose of this searching technique is to break up a page or piece into logical paragraphs. Paragraphs, in the WebL world, are longer regions of a page that logically belong together. Paragraphs in WebL should not be confused with HTML paragraphs (marked up with <*p*> ... </*p*> elements). Example paragraphs in WebL might be sequences of markup each terminated with a *br* tag, or the regions between a set of images. WebL allows the programmer to define his or own meaning of the term paragraph.

To allow the WebL programmer to define an own notion of paragraph, we introduce the notion of a *paragraph terminator.* A paragraph terminator is a tag which denotes the end of a paragraph. For example, the *br* tag might be denoted as a paragraph terminator. It is important to note that identifying a non-empty HTML element such as *font* as a terminator, signifies that both the begin tag <*font*> and end tag </*font*> are to regarded as paragraph terminators. Typically sets of terminators are used to break a page into paragraphs. For example, we can specify that all *br* and *p* tags are regarded as paragraph terminators, or that all tags except *i*, *b*, *font*, and *tt* are regarded as paragraph terminators.

Breaking a page into paragraphs with a specific set of paragraph terminators then proceeds as follows:

- Identify all the paragraph terminators on the page.
- Build a result piece set of paragraphs, namely all the regions that appear between successive terminators on the page (bounding terminator tags excluded). This involves the insertion of unnamed tags as placeholders.
- Remove from the result piece set all those pieces *p* that consist of white space only, i.e. applying *Markup(p)* returns a string containing only ‘ ‘, \n, \r, \t, and character 160 (character code of “ ”).

The paragraph search function *Para* expects a piece or page as first argument, and a specification of the paragraph terminators as the second argument. The function returns the pieceset of paragraphs. The paragraph terminator specification is in the form or a string of tag names, delimited by white space. For example:

```
var p = Para(page, "br p table li")
```

indicates the *br*, *p*, *table*, and *li* elements should be regarded as paragraph terminators.

Sometimes it is more convenient to specify the tags that should *not* be regarded as paragraph terminators. This is done by making the first element name in the paragraph terminator specification a "-":

```
var p = Para(page, "- font a b i tt img")
```

This indicates that all tags except for *font*, *a*, *b*, *i*, *tt*, and img should be regarded as paragraph terminators.

The last example illustrates a very useful application of the *Para* function. HTML distinguishes between *inline* elements and *block* elements. Block elements typically start and end on a fresh line in the displayed web page. Inline elements flow in the text stream and do not typically start or end a fresh line. Sometimes it is necessary to extract the blocks of inline elements, that make up the paragraphs of a Web page. As the number of inline HTML 4.0 elements are relatively small, we can accomplish this with the following WebL statement:

```
Para(page, "- tt i b u s strike big small em string
 dfn code samp kbd var cite acronym a img applet
 object font basefont script map q sub sup span
 bdo iframe input select textarea label button")
```

In a similar vein, the *Para* function can also play a role when extracting text from a Web page. This addresses a problem of the *Text* function when retrieving the text of a page. For example, applying the *Text* function to the following page

```
<li>word A</li><li>word B</li>
```

results in the text string "word Aword B", where two words unexpectedly flow together. To insert an extra space at the word boundary is dependent on whether a breaking tag is present or not. The problem can be solved with a script of the following form:

```
var P = Para(page,
    "- tt i b u s strike big small em string
    dfn code samp kbd var cite acronym a img applet
    object font basefont script map q sub sup span
    bdo iframe input select textarea label button");

var R = "";
every p in P do
    R = R + Text(p) + " ";
end
```

In conclusion, note that

```
Para(page, "-")
```

is nearly equivalent to

```
PCData(page)
```

except for the fact that pieces with no content are filtered out.

Filtering Pieces

Even though the piece searching functions introduced so far already provide powerful ways of extracting pieces from a web page, it might still not be enough. Suppose it is necessary to restrict the contents of a piece set to those elements whose attributes match some criteria. For example, we might be interested in all HTML anchors that point to a specific site. Exactly for this purpose the builtin *Select* function allows you to filter the contents of a piece set according to a selection function. (The *Select* function also supports filtering of sets and lists in a similar manner.). The following code fragment illustrates how the select function might be used in this case:

```
import Str;

var A = Select(
        Elem(P, "a"),
        fun(a) Str_StartsWith(a, "http://site.com") end
    )
```

Note how the selection function is passed as the second argument to *Select*. The *Select* function iterates over the elements of its first argument, repeatedly invoking the selection function to determine if that element should be included in the result piece set. The selection function must have a single formal argument and must return a boolean value that indicates whether its argument should be included in the result piece set or not. You are free to specify any selection criteria as long is the result of the function is of type boolean.

TABLE 17. Piece Set Searching Functions

Function	Description
Elem(p: page): pieceset	Returns all the elements in a page.
Elem(p: page, name: string): pieceset	Returns all the elements in page *p* with a specific *name*.
Elem(q: piece): pieceset	Returns all the elements that are contained (nested) in piece *q*.
Elem(q: piece, name: string): pieceset	Returns all the elements with a specific *name* contained in piece *q*.
Para(p: page, paraspec: string): pieceset	Extracts the paragraphs in *p* according to the paragraph terminator specification *paraspec*.
Para(p: piece, paraspec: string): pieceset	Extracts the paragraphs in *p* according to the paragraph terminator specification *paraspec*.
Pat(p: page, regexp: string): pieceset	Returns all the occurrences of a regular expression pattern in page *p*.
Pat(q: piece, regexp: string): pieceset	Returns all the occurrences of a regular expression pattern located inside the piece *q*.
PCData(p: page): pieceset	Returns the "parsed character data" of the page. This corresponds to the individual seqences of text on the page, as delimited by markup tags.
PCData(p: piece): pieceset	Returns the "parsed character data" of the piece. This corresponds to the individual seqences of text inside the piece, as delimited by markup tags.
Seq(p: page, pattern: string): pieceset	Matches all the occurrences of a sequence of elements identified by *pattern*. See "PCData search" on page 73.
Seq(p: piece, pattern: string): pieceset	Matches all the occurrences of a sequence of elements identified by *pattern* inside the piece *p*. See "PCData search" on page 73.

Miscellaneous Functions

The markup algebra includes several miscellaneous functions for converting between different value types, for example turning a string into a page and back, accessing the begin and end tags of a piece (See Table 18). To give a feeling for how these functions are used, we first define a new page containing a heading and 2-by-2 table:

```
var P = NewPage("<html><body>
    <h1>Test Page</h1>
    <table>
        <tr>
            <td align=center>A</td><td>100</td>
        </tr>
        <tr>
            <td align=center>B</td><td>230</td>
        </tr>
    </table>
    </body></html>", "text/html");
```

Note that the second argument to *NewPage* defines the parser to be used to parse the string into a page. For the definition above, the following WebL expressions evaluate in the following manner:

```
Markup(P)               // Returns "<html><body> ...
                        // ... </html>" as above.
var H = Elem(P, "h1")[0]   // Returns the first H1.
Markup(H)           // Returns "<h1>Test Page</h1>."

Text(P)             // Returns "Test Page A 100 B 230"
                    // (including white space).
Text(H)             // Returns "Test Page".

Name(H)             // Returns "h1".

var T = Elem(P, "td")   // Returns all the TD elements.
Markup(T[0])        // Returns
                    // "<td align="center">A</td>".
Markup(T[1])        // Returns "<td>100</td>".
Text(T[0])          // Returns "A".
Text(T[1])          // Returns "100".
Name(T[0])          // Returns "td".
```

```
T[0].align          // Returns "center".

var x = BeginTag(H), y = EndTag(H);

Page(H) == P        // Returns true.
Page(x) == P        // Returns true.
Page(y) == P        // Returns true
```

The *Pretty* function is similar to the *Markup* function except that it pretty-prints the markup by indenting elements according to their nesting level. This is useful to study the structure of badly formatted HTML and XML pages. Note that pretty-printing a page involves a reformatting of white spaces and new lines, so the resulting string might differ dramatically from the original page source (sometimes enough to break scripts that worked correctly on the "ugly" page)[1].

TABLE 18. Miscellaneous Functions

Function	Description
BeginTag(q: piece): tag	Returns the begin tag of a piece.
EndTag(q: piece): tag	Returns the end tag of a piece.
ExpandCharEntities(p: page, s: string): string	Expands the character entities (eg. <, &) in *s* to their Unicode character equivalents. The DTD of page *p* is used for the lookups.
ExpandCharEntities(s: string): string	Expands the character entities (eg. <, &) in *s* to their Unicode character equivalents. The HTML 4.0 DTD is used for the lookups.
Markup(p: page): string	Turns a page object back into a string.
Markup(q: piece): string	Turns a piece object back into a string.
Name(q: piece): string	Returns the name of a piece, or the empty string in the case of *q* being unnamed.
NewPage(s: string, mimetype: string): page	Parses the string s with the mime-type indicated markup parser and returns a page object.
NewPiece(s: string, mimetype: string): piece	Equivalent to *Content(NewPage(s, mimetype))*.
NewPieceSet(s: set): pieceset	Converts a set of pieces into a piece set. Thows an *EmptySet* exception should *s* be empty.
NewPieceSet(p: page): pieceset	Returns an empty pieceset associated with with page *p*.
Page(q: piece): page	Returns the page a piece belongs to.
Page(t: tag): page	Returns the page a tag belongs to.
Pretty(p: page): string	Returns a pretty-printed version of the page.

1. WebL tries to ensure that the pretty-printed page still renders in the browser in the same manner as the original page by using some limited inbuilt knowledge about markup. For example, HTML preformatted elements (PRE) are not changed.

TABLE 18. Miscellaneous Functions

Function	Description
Pretty(q: piece): string	Returns a pretty-printed version of a piece.
Size(p: pieceset):int	Returns the number of pieces belonging to *p*.
Text(p: page): string	Returns the text (sans tags) of a page.
Text(q: piece): string	Returns the text (sans tags) of a piece.

Piece Comparison

Pieces can be compared for equality, containment, position relative to each other, and so on. These tests play a very important role in the piece set operators introduced in the following section.

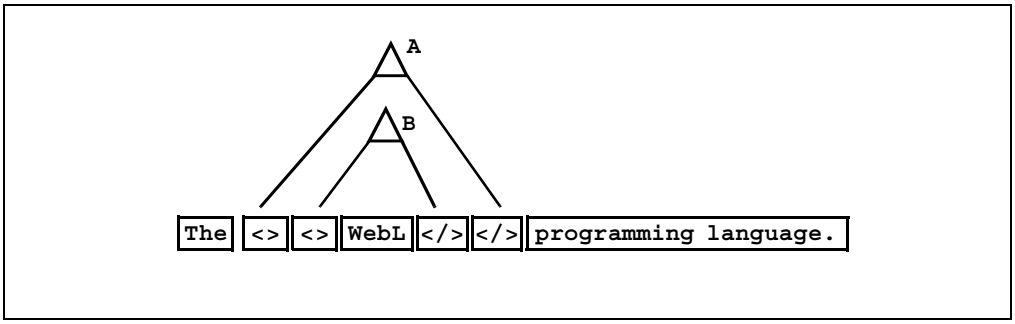
Without regard to unnamed pieces, the comparison of (named) pieces is quite straightforward — for example, piece *x* is equal to piece *y* if the following is true:

```
BeginTag(x) == BeginTag(y) and EndTag(x) == EndTag(y)
```

Unfortunately, the situation is more complicated when unnamed pieces are involved. (So far we have only seen unnamed pieces being created as a side-effect of the *Pat* function — following sections will illustrate that many other functions have a similar effect.) The problem is that two different pieces (according to our definition above) might have equivalent markup, which confuses the difference between the two pieces. This is a side effect of an unnamed tag becoming "invisible" when the piece is converted to markup.

For example, in Figure 4, piece *B* is nested inside piece *A*. Applying the *Markup* function to *A* and *B* strips away the unnamed pieces to return the string "WebL" (without any markup). Because of our handling of unnamed pieces as invisible entities (the place holders for patterns), piece *A* and *B* should be equal to each other from the programmer's point of view, but is not according to our earlier definition.

FIGURE 4. Nested Unnamed Pieces



The first intuition is that WebL should merge neigbouring unnamed tags, so that the equality problem goes away. Unfortunately, experience has shown that merging of unnamed tags is a bad idea. Without going into too much detail, merging of unnamed tags complicates the programmer's mental understanding of the current "shape" of the page, as merging might happen at unexpected situations. This often causes problems when a page is subsequently modified. To give a flavor of the problems that might occur, suppose piece *A* of the figure was created by thread *A*, and piece *B* of the figure was created by an independent thread *B*. Now let's suppose thread *A* inserts a character 'x' directly after the begin tag of *A*. In the case of separate (i.e. non-merged) unnamed tags, the resulting situation is easy to visualize. However, with merged unnamed tags, thread A will insert the character inside the piece *B* created by thread *B*, which might be unexpected by thread B. These type of problems caused us to reject unnamed tag merging.

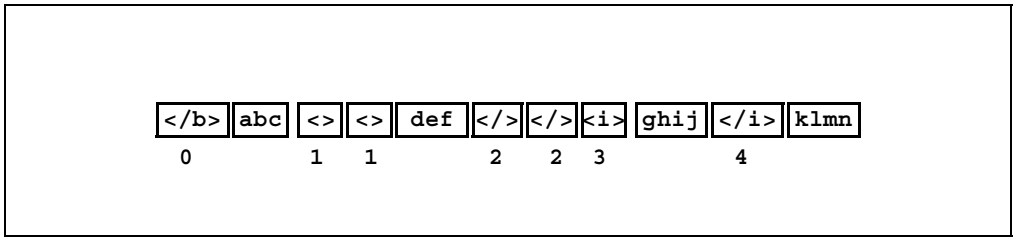
Instead, to ensure that A and B are equal, WebL introduces the concept of *positions*. The position of a tag is a numerical rank of the tag in a page. We number tags from 0 onwards in the order of occurrence in the page, all the while ensuring that sequences of unnamed tags have the same number[1]. Figure 5 shows the position numbering for a more complicated page consisting of named and unnamed tags. Comparisons of pieces is then made according to the positions of the begin and end tags of the pieces. For example, our definition of piece equality of *x* and *y* becomes[2]:

1. Readers concerned about inefficient renumbering of tag positions after inserting or deleting tags should be aware that behind the scene, WebL uses an efficient encoding that prevents renumbering positions for large parts of the page after a modification is performed.

```
pos(BeginTag(x)) == pos(BeginTag(y)) and
pos(EndTag(x)) == pos(EndTag(y))
```

Using the notion of positions, we can thus define equality, containment, etc. as in Table 19. In this table we use the notation *beg* to indicate the position of the begin tag of a piece, and *end* to indicate the position of the end tag of a piece. Note that the piece comparison operators *equal*, *inside*, *after*, etc. are not defined in the WebL language itself — the following section will introduce new language operators based on these definitions.

FIGURE 5. Example of Position Numbering



2. We introduce here a fictitious WebL function called *pos* that returns the numerical position of a tag value.

TABLE 19. Comparing Pieces x and y

Relationship between x and y	Definition
x ***equal*** y	$beg(x) = beg(y) \wedge end(x) = end(y)$
x ***inside*** y	$beg(y) \leq beg(x) \wedge end(x) \leq end(y) \wedge \neg(beg(x) = beg(y) \wedge end(x) = end(y))$
x ***contain*** y	$beg(x) \leq beg(y) \wedge end(y) \leq end(x) \wedge \neg(beg(x) = beg(y) \wedge end(x) = end(y))$
x ***after*** y	$end(y) < beg(x)$
x ***before*** y	$end(x) < beg(y)$
x ***overlap*** y	$beg(x) \leq end(y) \wedge beg(y) \leq end(x) \wedge \neg(beg(x) = beg(y) \wedge end(x) = end(y))$

Piece Set Operators and Functions

All the piece set operators are summarized in Table 20 on page 101. Note that all piece set operators accept both pieces and piece sets as operands. Piece operands are converted automatically to a piece set with the operand as the only element. Most of the operators have a formal definition as defined in Table 22 on page 105. The remainder of this section attempts to give an intuitive explanation of the operators with the help of examples. In our examples *X* will denote a page, *P* and *Q* will denote piece sets, and *p* and *q* will denote elements of *P* and *Q* respectively.

I. Basic Operators

Basic piece set manipulation includes the set union, intersection, and exclusion operators.

Set Union (P + Q). The set union operator + merges two piece sets into a single piece set, and eliminates duplicate pieces from the result. Example:

```
// Retrieve level 1 and two headings from a page
Elem(X, "h1") + Elem(X, "h2")
```

Set Exclusion (P - Q). The set exclusion operator "-" removes all pieces from the left operand that are elements of the right operand. Example:

```
// Retrieve all level 1 headings except for those
// that contain the word "Figure".
Elem(X, "h1") -
(Elem(X, "h1") contain Pat(X, "Figure"))
```

Set Intersection (P * Q). The set intersection * computes the intersection between its operands. Example:

```
// Retrieve all the occurrences of the word "WebL"
// written in bold and in italic.
(Pat(X, "WebL") inside Elem(X, "b")) *
(Pat(X, "WebL") inside Elem(X, "i"))
```

II. Positional Operators

Positional operators express relationships between pieces according to their order in a page. Most positional operators have a negated or inverted version that is indicated by an operator symbol written with an exclamation point (!).

Indexing P[i]. The index operator [] extracts the nth element of a piece set *P*. Pieces are numbered from 0 to Size(P) - 1. Examples:

```
// Extract the 4'th table from a page.
Elem(X, "table") [4]

// Extract the 2'nd row of the 3'rd table.
Elem(Elem(X, "table")[3], "tr")[2]

// Extract the 2'nd row of the table containing the
// word WebL
var t = Elem(X, "table") contain Pat(X, "Webl");
(Elem(X, "tr") inside t)[2]
```

P before/!before Q. The *before* operator returns all the elements of *P* that are before (or not before) any element of *Q*. Note that this is equivalent to all the elements of *P* that are before (or not before) the *last* element of *Q*. Consequently, we often need to index into *Q* to reduce it to a single piece. Examples:

```
// Retrieve all the H2's before the appendix
// (assuming only a single appendix is present).
Elem(X, "h2") before
(Elem(X, "h1") contain Pat(X, "Appendix"))

// Retrieve all the headings from Chapter 4 onwards.
Elem(X, "h1") !before
(Elem(X, "h1") contain Pat(X, "Chapter 4"))

// Retrieve all the italic elements except the last.
Elem(X, "i") before Elem(X, "i")

// Retrieve the last italic element.
Elem(X, "i") !before Elem(X, "i")
```

P directlybefore/!directlybefore Q. The *directlybefore* operator returns the pieces of *P* that are directly before (or not directly before) any element of *Q*. A piece *p* of *P* is directly before a piece *q* of *Q* if no other piece in *P* appears between *p* and *q*. For example, given page *X* contains (excluding the line numbers on the left):

```
1       <h1>A</h1>
2           <i>a</i>
3           <i>b</i>
4           <b>c</b>
5       <h1>B</h1>
6           <i>d</i>
7           <i>e</i>
8       <h1>C</h1>
9           <i>f</i>
10          <i>g</i>
11      <h1>D</h1>
12          <i>h</i>
13          <i>i</i>
14      <h1>E</h1>
```

we can compute the following:

```
// Retrieve the italics directly before H1's,
// i.e. lines 3, 7, 10, 13.
Elem(X, "i") directlybefore Elem(X, "h1")

// Retrieve the italics not directly before H1's,
// i.e. lines 2, 6, 9, 12.
Elem(X, "i") !directlybefore Elem(X, "h1")

// Retrieve all elements directly before H1's,
// i.e. lines 4, 7, 10, 13.
Elem(X) directlybefore Elem(X, "h1")

// Retrieve the second element directly before H1's,
// i.e. lines 3, 6, 9, 12.
Elem(X) directlybefore
(Elem(X) directlybefore Elem(X, "h1"))
```

P after/!after Q. The *after* operator returns all the elements of *P* that are after (or not after) any element of *Q*. Note that this is equivalent to all the elements of *P* that are after (or not after) the *first* element of *Q*. Consequently, we often need to index into Q to reduce it to a single piece. Examples:

```
// Retrieve all the H2's after the appendix
// (assuming only a single appendix is present).
Elem(X, "h2") after
(Elem(X, "h1") contain Pat(X, "Appendix"))

// Retrieve all the headings before Chapter 4
// inclusive.
Elem(X, "h1") !after
(Elem(X, "h1") contain Pat(X, "Chapter 4"))

// Retrieve all the italic elements except the last.
Elem(X, "i") before Elem(X, "i")

// Retrieve the last italic element.
Elem(X, "i") !before Elem(X, "i")
```

P directlyafter/!directlyafter Q. The *directlyafter* operator returns the pieces of *P* that are directly after (or not directly after) any element of *Q*. A piece *p* of *P* is directly after a piece *q* of *Q* if no other piece in *P* appears between *p* and *q*. Examples (based on the previous page object *X*):

```
// Retrieve the italics directly after H1's,
// i.e. lines 2, 6, 9, 12.
Elem(X, "i") directlyafter Elem(X, "h1")

// Retrieve the italics not directly after H1's,
// i.e. lines 3, 4, 7, 10, 12.
Elem(X, "i") !directlyafter Elem(X, "h1")

// Retrieve all elements directly after H1's,
// i.e. lines 2, 6, 9, 12.
Elem(X) directlyafter Elem(X, "h1")

// Retrieve the second element directly after H1's,
// i.e. lines 3, 7, 10, 13.
Elem(X) directlyafter
(Elem(X) directlyafter Elem(X, "h1"))
```

P overlap/!overlap Q. The *overlap* operator returns the pieces of *P* that overlap (or do not overlap) any element of *Q*. Example:

```
// Find all the occurrences of words that are
// italic or partially consists of italic text.
Pat(X, '\w+') overlap Elem(X, "i")
```

III. Hierarchical Operators

The hierarchical operators express relationships between pieces involving their hierarchical nesting in the element parse tree.

P inside/!inside Q. The *inside* operator returns the pieces of *P* that are nested inside (or not nested inside) any piece of *Q*. Examples:

```
// Retrieve all the rows in the third table.
Elem(X, "tr") inside Elem(X, "table")[3]

// Retrieve all the italic elements not in a table.
Elem(X, "i") !inside Elem(X, "table")
```

P contain/!contain Q. The *contain* operator returns the pieces of *P* that contain (or do not contain) any piece of *Q*. Examples:

```
// Retrieve all the level 2 headings with
// italic characters.
Elem(X, "h2") contain Elem(X, "i")

// Retrieve all the tables that mention "program".
Elem(X, "table") contain Pat(X, "program")
```

P directlyinside/!directlyinside Q. The *directlyinside* operator returns all the elements of *P* that are inside (or not inside) any element of *Q*, and in addition are not inside another element of *P*. Intuitively this retrieves the "outermost" element of all nested elements. Given a page of the following form:

```
1   <UL>
2       <LI>First Section</LI>
3       <LI>Second Section</LI>
4       <LI>Third Section
5           <UL>
6               <LI>First Subsection</LI>
7               <LI>Second Subsection</LI>
8           </UL>
9       </LI>
10      <LI>Fourth Section</LI>
11  </UL>
```

we can calculate the following:

```
// All the list items in lists,
// i.e. lines 2, 3, 4, 6, 7, 9.
Elem(X, "li") inside Elem(X, "ul")

// All the list items in the first list,
// i.e. the elements on lines 2, 3, 4-9, 6, 7, 10.
Elem(X, "li") inside Elem(X, "ul")[0]

// All the items directly in the first list,
// i.e. lines 2, 3, 4-9, 10.
Elem(X, "li") directlyinside Elem(X, "ul")[0]

// Outermost items in the first list,
// i.e. lines 2, 3, 4-9, 10.
var x = Elem(X, "li") inside Elem(X, "ul")[0];
x !inside x
```

P directlycontain/!directlycontain Q. The *directlycontain* operator returns all the elements of *P* that contain (or do not contain) any element of *Q*, and in addition do not contain another element of *P*. Intuitively this retrieves the "innermost" element of all nested elements. Given the page defined previously, we can calculate:

```
// The lists that contain the first subsection,
// i.e. elements on lines 1-11, 5-8.
Elem(X, "ul") contain Pat(X, "First Subsection")

// The list that directly contains the first subsection
// i.e. element in lines 5-8.
Elem(X, "ul") directlycontain
Pat(X, "First Subsection")

// Innermost list that containsthe first subsection,
// i.e. element in lines 5-8.
var x = Elem(X, "ul") contain
   Pat(X, "First Subsection");
x !contain x
```

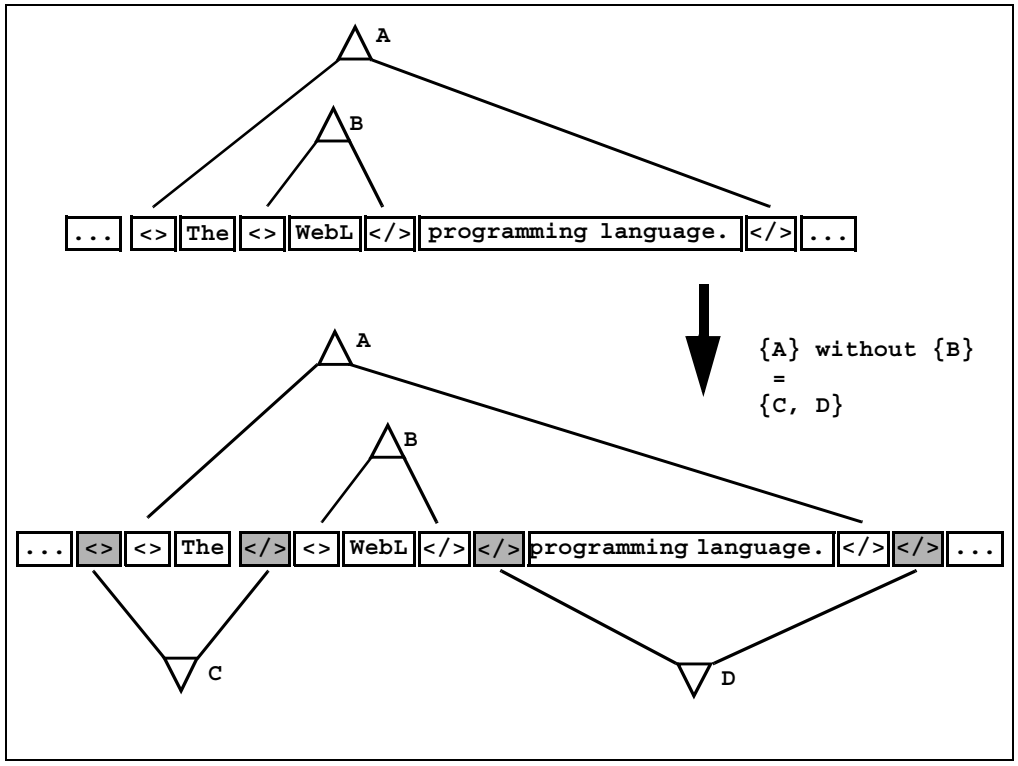
IV. Regional Operators

The regional operators construct new pieces to identify parts of a page. (Many other operators return pieces that existed only before the operator was applied.)

P without Q. The without operator returns the pieces of *P* where parts of *Q* that overlap with a piece in *P* are "cut" away. This might involve creating several new pieces from a piece of *P* and inserting new unnamed tags as necessary. Figure 6 gives an example where the word WebL is removed from a sentence. Note how unnamed tags are inserted to the left and right of piece *A*. Examples:

```
// "Cut" up the second table into its
// constituent lines.
Elem(X, "table")[1] without Pat(X, '\n')

// Remove all the bold text from
// the first paragraph.
Elem(X, "p")[0] without Elem(X, "b")
```

FIGURE 6. Operation of P without Q

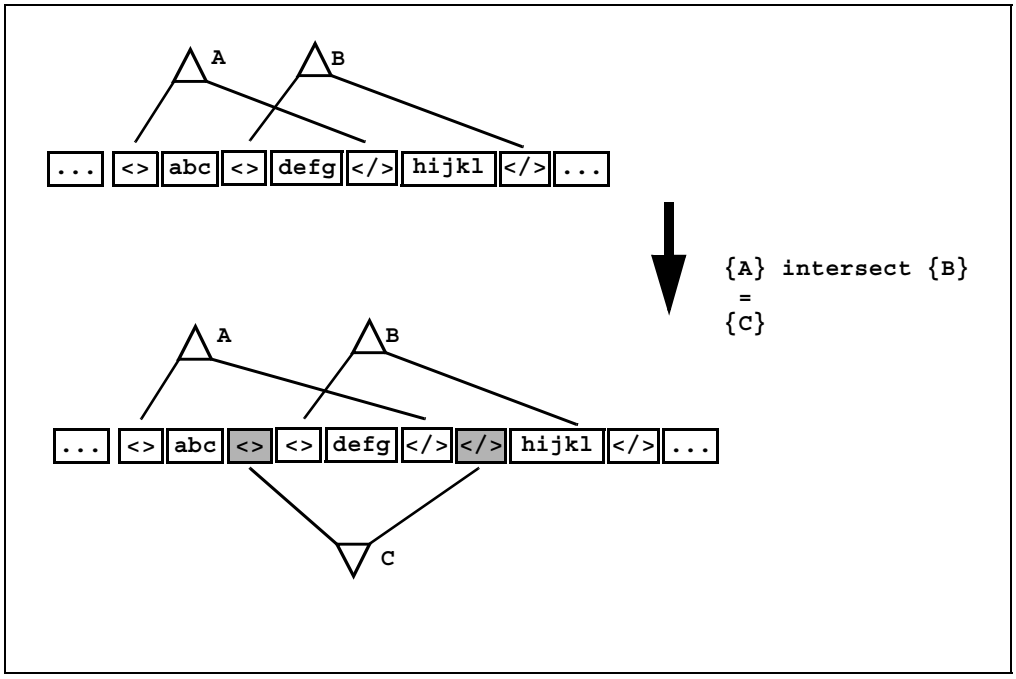


P intersect Q. The *intersect* operator intersects each element of *P* with all the overlapping pieces of *Q*. The resulting piece set contains all the parts of *P* that are in common with pieces of *Q*. As parts of pieces of *P* are cut away by the intersection, new pieces need to be created, and thus new unnamed tags are inserted into the page. Another way of thinking about the operator is that it calculates the overlap between pieces. Figure 7 shows how this is done.

Example:

```
// The parts of a page that is both italic and bold.
Elem(X, "i") intersect Elem(X, "b")
```

FIGURE 7. Operation of P intersect Q



V. Miscellaneous Functions

Children(p). The *Children* function returns all the children pieces of piece *p*. The children of a piece include all the elements directly contained in the piece and all the text segments directly contained in the piece. Markup elements that are only parially inside *p* because of overlap, are not regarded as children of *p*. For example, the children of the following TD element consisting of nested *I* and *B* elements:

```
<td>abc<i><b>def</b></i>ghi<b>jkl</b>mno</td>
```

are the pieces represented by:

```
"abc", "<i><b>def</b></i>",
"ghi", "<b>jkl</b>", "mno"
```

Examples:

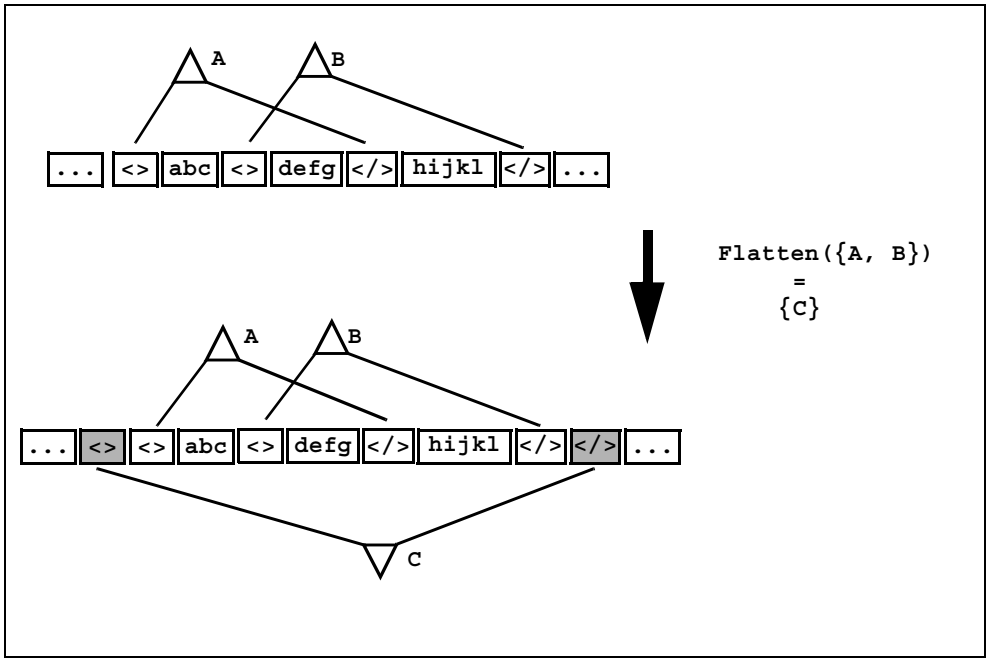
```
// Everything inside the first table.
Children(Elem(X, "table")[0])

// Program to walk recursively through a page.
var walk = fun(x)
   if Name(x) != "" then      // Named piece
      every p in Children(x) do
         walk(p)
      end
   else
      PrintLn(Text(x), " parent=", Name(Parent(x)))
   end
end;
var P = NewPage("<td>abc<i><b>def</b></i>
   ghi<b>jkl</b>mno</td>", "text/xml");
walk(Elem(X, "td")[0])
```

Parent(p). The *Parent* function returns the direct parent (enclosing) element of piece *p*. It is implemented by looking at named tags *t* from right to left starting just before the left tag of *p*, identifying the piece *q* that tag *t* belongs to, and determining if the corresponding end tag of *q* follows the end tag of *p*. Example:

```
// Locate the Parent element of the second table.
Parent(Elem(P, "table")[1])
```

Flatten(P). The *Flatten* function returns the union of all elements of *P*. Intuitively two overlapping pieces *p* and *q* of *P* are replaced repeatedly by a single "joined" piece that covers the union of the regions *p* and *q* covered. This also has the effect of removing nested elements of *P*. New unnamed pieces are inserted into the page to create these new pieces. Figure 8 shows how two overlapping pieces are flattened.

FIGURE 8. Flattening a Piece Set



Content(p). The *Content* function returns the content of piece *p*. The content of a piece is the part of the page between the begin and end tag of *p* (exclusive). The *Content* function can also be applied to a page object, in which case a piece is returned that starts at the beginning of the page and ends at the end of the page. In both cases, new unnamed tags are inserted into the page (Figure 9). For example, given a page:

```
<td>abc<i>def</i></td>
```

we can calculate the following:

```
// Content of the TD element,
// i.e. "abc<i>def</i>".
Content(Elem(P, "td")[0])

// Content of the whole page,
// i.e. "<td>abc<i>def</i></td>".
Content(P)
```

FIGURE 9. Application of the Content Function

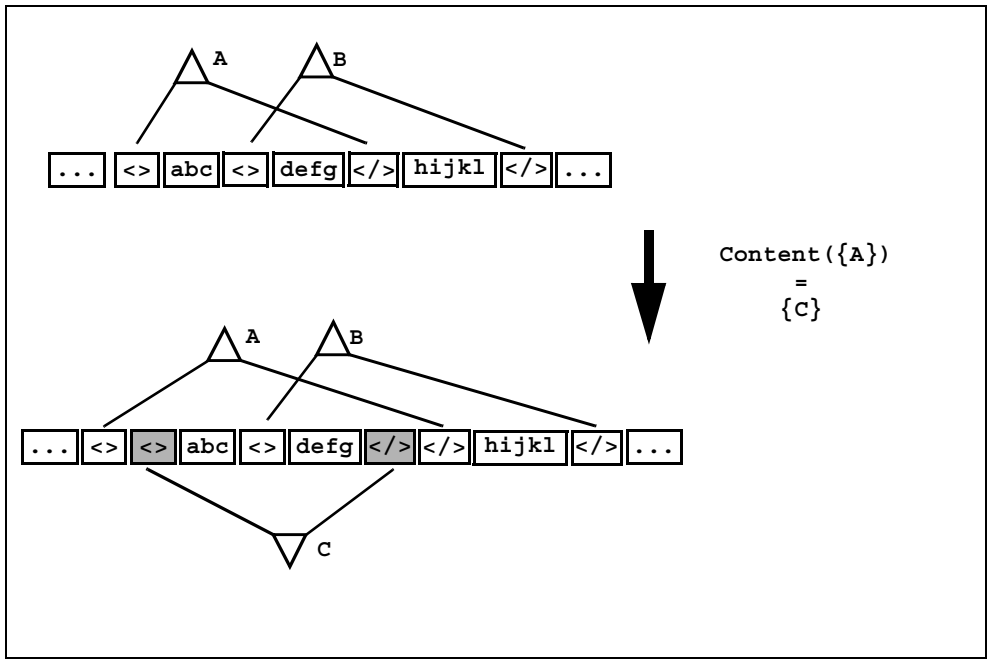


TABLE 20. Piece and Piece Set Operators

Function	Description
+(q1: piece, q2: piece): pieceset +(q: piece, s: pieceset): pieceset +(s: pieceset, q: piece): pieceset +(s1: pieceset, s2: pieceset): pieceset	Piece set union.
-(q1: piece, q2: piece): pieceset -(q: piece, s: pieceset): pieceset -(s: pieceset, q: piece): pieceset -(s1: pieceset, s2: pieceset): pieceset	Piece set difference.
*(q1: piece, q2: piece): pieceset *(q: piece, s: pieceset): pieceset *(s: pieceset, q: piece): pieceset *(s1: pieceset, s2: pieceset): pieceset	Piece set intersection.
[](s: pieceset, i: int): piece	Indexing into a piece set. Pieces are numbered *0* to *Size* - 1.
inside(p: piece, q: piece): pieceset **inside**(p: pieceset, q: piece): pieceset **inside**(p: piece, q: pieceset): pieceset **inside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are located inside any element of *q*.
!**inside**(p: piece, q: piece): pieceset !**inside**(p: pieceset, q: piece): pieceset !**inside**(p: piece, q: pieceset): pieceset !**inside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not located inside any element of *q*.
directlyinside(p: piece, q: piece): pieceset **directlyinside**(p: pieceset, q: piece): pieceset **directlyinside**(p: piece, q: pieceset): pieceset **directlyinside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are directly inside any element of *q*.
!directlyinside(p: piece, q: piece): pieceset **!directlyinside**(p: pieceset, q: piece): pieceset **!directlyinside**(p: piece, q: pieceset): pieceset **!directlyinside**(p: pieceset, p: pieceset): pieceset	All the elements of *p* that are not directly inside any element of *q*.
contain(p: piece, q: piece): pieceset **contain**(p: pieceset, q: piece): pieceset **contain**(p: piece, q: pieceset): pieceset **contain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that contain any element of *q*.
!contain(p: piece, q: piece): pieceset **!contain**(p: pieceset, q: piece): pieceset **!contain**(p: piece, q: pieceset): pieceset **!contain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not contain any element of *q*.

TABLE 20. Piece and Piece Set Operators

Function	Description
directlycontain(p: piece, q: piece): pieceset **directlycontain**(p: pieceset, q: piece): pieceset **directlycontain**(p: piece, p: pieceset): pieceset **directlycontain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that directly contain any element of *q*.
!directlycontain(p: piece, q: piece): pieceset **!directlycontain**(p: pieceset, q: piece): pieceset **!directlycontain**(p: piece, q: pieceset): pieceset **!directlycontain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not directly contain any element of *q*.
after(p: piece, q: piece): pieceset **after**(p: pieceset, q: piece): pieceset **after**(p: piece, q: pieceset): pieceset **after**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are after any element of *q*.
!after(p: piece, q: piece): pieceset **!after**(p: pieceset, q: piece): pieceset **!after**(p: piece, q: pieceset): pieceset **!after**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not after any element of *q*.
directlyafter(p: piece, q: piece): pieceset **directlyafter**(p: pieceset, q: piece): pieceset **directlyafter**(p: piece, q: pieceset): pieceset **directlyafter**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that follow directly after any element of *q*.
!directlyafter(p: piece, q: piece): pieceset **!directlyafter**(p: pieceset, q: piece): pieceset **!directlyafter**(p: piece, q: pieceset): pieceset **!directlyafter**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not follow directly after any element of *q*.
before(p: piece, q: piece): pieceset **before**(p: pieceset, q: piece): pieceset **before**(p: piece, q: pieceset): pieceset **before**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that precede any element of *q*.
!before(p: piece, q: piece): pieceset **!before**(p: pieceset, q: piece): pieceset **!before**(p: piece, q: pieceset): pieceset **!before**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not precede any element of *q*.
directlybefore(p: piece, q: piece): pieceset **directlybefore**(p: pieceset, q: piece): pieceset **directlybefore**(p: piece, q: pieceset): pieceset **directlybefore**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are directly before any element of *q*.

TABLE 20. Piece and Piece Set Operators

Function	Description
!directlybefore(p: piece, q: piece): pieceset **!directlybefore**(p: pieceset, q: piece): pieceset **!directlybefore**(p: piece, q: pieceset): pieceset **!directlybefore**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not directly before any element of *q*.
overlap(p: piece, q: piece): pieceset **overlap**(p: pieceset, q: piece): pieceset **overlap**(p: piece, q: pieceset): pieceset **overlap**(p: pieceset, p: pieceset): pieceset	All the elements of *p* that overlap any element in *q*.
!overlap(p: piece, q: piece): pieceset **!overlap**(p: pieceset, q: piece): pieceset **!overlap**(p: piece, q: pieceset): pieceset **!overlap**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not overlap any element in *q*.
without(p: piece, q: piece): pieceset **without**(p: pieceset, q: piece): pieceset **without**(p: piece, q: pieceset): pieceset **without**(p: pieceset, q: pieceset): pieceset	All the elements of *p* where overlap with any element of *q* has been removed.
intersect(p: piece, q: piece): pieceset **intersect**(p: pieceset, q: piece): pieceset **intersect**(p: piece, q: pieceset): pieceset **intersect**(q: pieceset, p: pieceset): pieceset	All the elements of *p* that overlap an element in q, each of them repeatedly intersected with all overlapping elements in *q*.

TABLE 21. Piece and Piece Set Functions

Function	Description
Children(q: piece): pieceset	Returns a piece set consisting of all the direct children elements of *q* in the markup parse tree, unioned with pieces representing all the text segments in *q* (without all the nested text segments).
Parent(q: piece): piece	Returns the element in which *q* is nested (direct parent in the parse tree).
Flatten(s: pieceset): pieceset	Returns a "flattened" piece set (without any overlappings) of all the parts of the page that piece set s covers.
Content(p: page): piece	Returns a piece that encompasses the whole page *p*.
Content(q: piece): piece	Returns a piece inside *q*, representing everything that is inside *q* excluding the begin tag and end tag of *q*.

TABLE 22. Formal Definitions of Piece Set Operators

Operator	Definition
P + Q	$P \cup \{ q \in Q \mid \neg\exists p\ p \in P \wedge p$ ***equal*** $q \}$
P - Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***equal*** $q \}$
P * Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***equal*** $q \}$
P inside Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***inside*** $q \}$
P !inside Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***inside*** $q \}$
P directlyinside Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***inside*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***inside*** $q \wedge p$ ***inside*** $r) \}$
P !directlyinside Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***inside*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***inside*** $q \wedge p$ ***inside*** $r) \}$
P contain Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***contain*** $q \}$
P !contain Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***contain*** $q \}$
P directlycontain Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***contain*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***contain*** $q \wedge p$ ***contain*** $r) \}$
P !directlycontain Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***contain*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***contain*** $q \wedge p$ ***contain*** $r) \}$
P after Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***after*** $q \}$
P !after Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***after*** $q \}$
P directlyafter Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***after*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***after*** $q \wedge p$ ***after*** $r) \}$
P !directlyafter Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***after*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***after*** $q \wedge p$ ***after*** $r)\}$
P before Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***before*** $q \}$
P !before Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***before*** $q \}$
P directlybefore Q	$\{ p \in P \mid \exists q\ q \in Q \wedge p$ ***before*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***before*** $q \wedge p$ ***before*** $r) \}$
P !directlybefore Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***before*** $q \wedge$ $(\neg\exists r\ r \in P \wedge r$ ***before*** $q \wedge p$ ***before*** $r) \}$
P overlap Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***overlap*** $q \}$
P !overlap Q	$\{ p \in P \mid \neg\exists q\ q \in Q \wedge p$ ***overlap*** $q \}$

Page Modification

Page modification is an important part of the WebL markup algebra. As we have seen already, the attributes of markup elements can be elegantly modified by accessing the fields of pieces. This section will focus on how to insert pieces into a page, delete pieces from a piece, and replace pieces of a page.

Creating Pieces

There are several ways to create new pieces (See Table 23). After a new piece has been created, it can be inserted into a page at a specific position. We already introduced the *NewPage* function which takes a string and a mimetype as argument, and returns a page object. We also know that then applying the *Content* function to a page returns a piece covering the whole page. In fact, the code to create a piece in this manner:

```
Content(NewPage("<html> ... </html>", "text/html"))
```

occurs so often that we also use the following short-hand:

```
NewPiece("<html> ... </html>", "text/html")
```

Another way of creating a new piece is to pass the begin tag and end tag of two arbitrary pieces to the *NewPiece* function (Figure 10). The function returns a new unnamed piece with new unnamed tags inserted just before and after the begin tag and end tag respectively (to "wrap" its contents). The *NewPiece* function will also wrap a piece argument in the same manner[1].

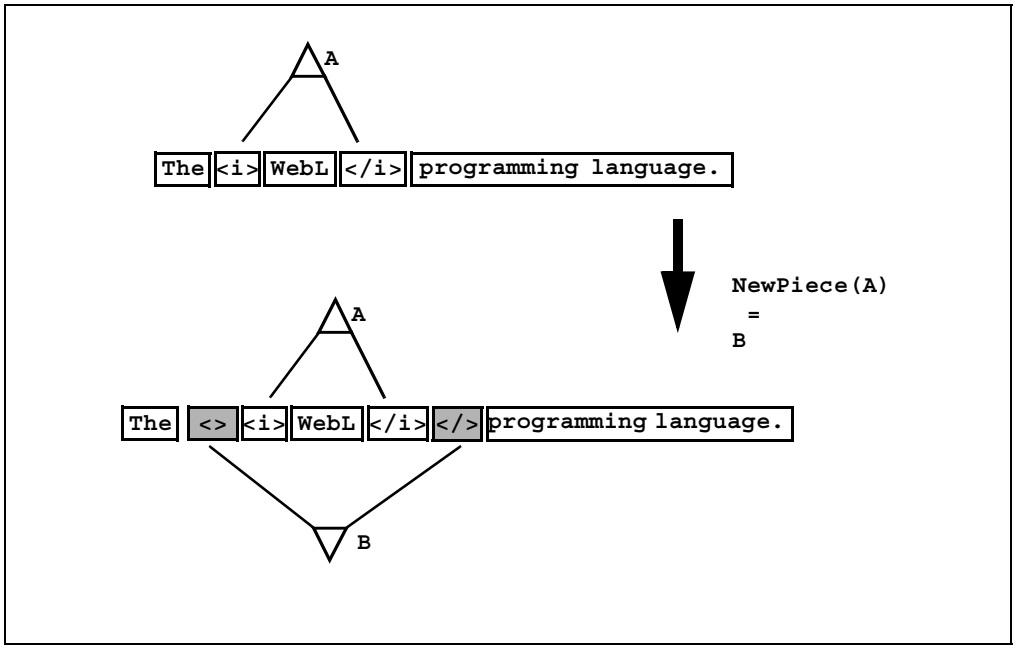
The *NewNamedPiece* function works in a similar manner as *NewPiece*, except that a new piece with the indicated name is created. Seeing that any begin and end tag pair (not belonging to the same piece) can be passed to this function, programmers should be aware that invalid HTML or XML can be created where elements do not nest properly. As WebL uses a flexible internal page representation, the presence of overlapping elements does not present any problems.

1. Technically the function does not modify the contents of a page because only unnamed tags are inserted into the page.

Examples:

```
// Turn the text from the word "WebL" to the end of
// the sixth paragraph to italic.
var a = Pat(P, "WebL")[0], b = Elem(P, "p")[5];
NewNamedPiece("i", BeginTag(a), EndTag(b))

// Turn all occurrences of WWW to a hyperlink.
every x in Pat(P, "WWW") do
    var p = NewNamedPiece("a", x);
    p.href := "http://www.w3.org"
end
```

FIGURE 10. Application of the NewPiece function



Inserting Pieces

The functions *InsertBefore* and *InsertAfter* insert a piece into a page either before or after a specified tag. Inserting a piece *p* involves *copying* the contents of the piece and inserting the copied tags and text segments one after another at the destination point according to the following rules:

- All the text segments contained in *p* are copied.
- All the named tags contained in *p* are copied (also includes the named tags of *p* itself). Also suppose there exists a piece *q* that is either inside *p* or overlaps with *p.* In case *q* is inside *p*, both the begin tag and end tag of *q* will be copied to the destination. Otherwise, if *q* overlaps with *p* (and is not inside *p*), we will, according to our definition, only copy either the begin tag or end tag of *q*. To prevent this unfortunate situation with dangling pieces, the tag of *q* outside of *p* is also copied. In case of many dangling tags outside of *p*, we copy all of them, making sure that their relative ordering is preserved.
- Unnamed tags contained in *p* are *not* copied.

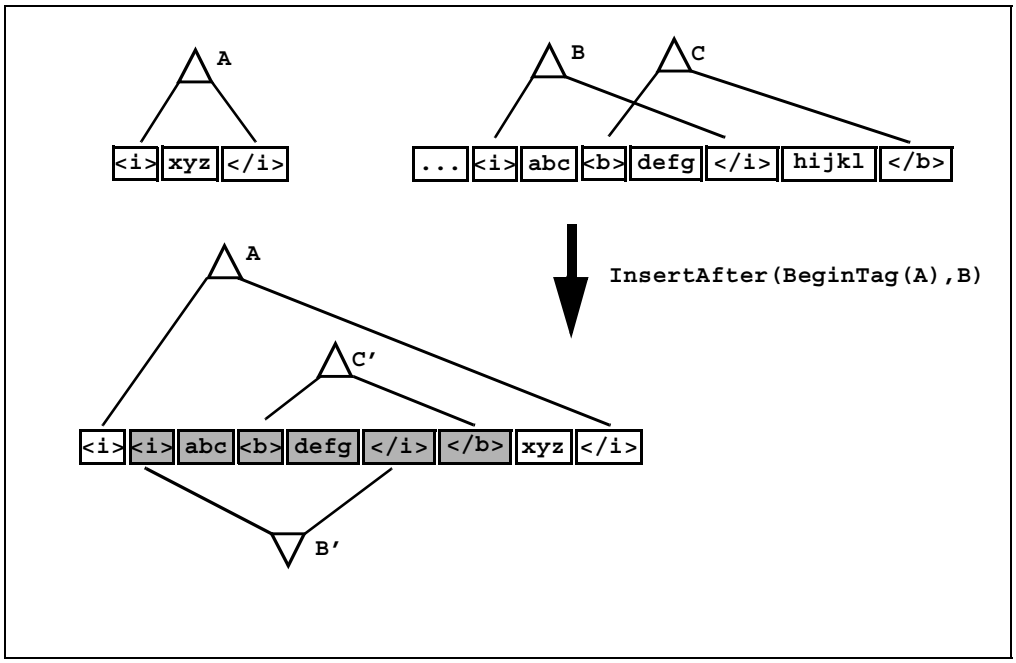
Figure 11 shows how copying piece *B* with an overlapping piece *C* after the begin tag of *A*, results in the copies *B'* and *C'* in the page. Note that the source page on the right top corner of the figure remains unchanged.

In case a piece set (instead of a piece) is passed to these two functions, each of the elements of the argument will be copied in sequence to the destination insertion point. Note that when the piece set contains nested elements, the nested elements will be inserted *twice or more times*, possibly one after another in the destination page.

Example:

```
// Insert an image at the beginning of each h1 tag.
var p = NewPiece("<img scr=a.gif>", "text/html");
every x in Elem(P, "h1") do
   InsertAfter(BeginTag(x), p)
end
```

FIGURE 11. **Copying Pieces during Inserts**



Deleting Pieces

The *Delete* function deletes a piece from a page. In case the function is passed a piece set argument, each of the elements of the argument piece is deleted.

One of the problems we face with deletion is that some program variables might still refer to pieces that were previously deleted. Accordingly, accessing these deleted pieces through these variables might cause some problems. To simplify the problem, we define the following sematics for deletion of a piece *q*:

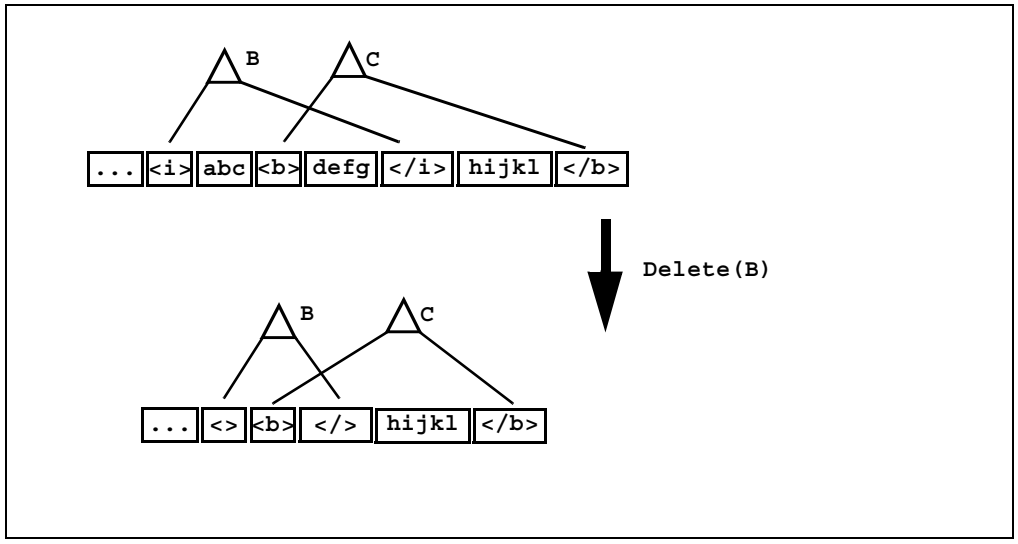
- All the text segments contained in piece *q* are physically removed from the page. (This is not a problem seeing that we cannot refer to text segments in the WebL markup algebra.)
- Unnamed tags inside *q* are left untouched.
- The tags of named pieces completely inside *p* are converted to unnamed tags. They still can be referred to but essentially become invisible.
- If *q* is named, then its tags are converted to unnamed tags.
- The tags of named pieces that overlap *q* (but are not inside *q*) are left untouched.

Figure 12 illustrates the situation when overlaps occur during deletions. Note how piece *C* remains named because its end tag is located outside and to the right of *B*. Of course, as we simply leave tags where they are in the page, we can imagine situations where the page fills up with unused tags after several deletions. To counter this problem, a *scrubber* process is periodically invoked to remove unused tags from a page (i.e. tags that are not accessible to the programmer as detected by the Java garbage collector).

Examples:

```
// Delete all occurrences of the word "cool".
Delete(Pat(P, "cool "))

// Remove all H1 and H2 headings.
Delete(Elem(P, "h1") + Elem(P, "h2"))
```

FIGURE 12. Deleting Pieces



Replacing Pieces

The *Replace* function deletes each piece in the first argument and inserts copies of the pieces of the second argument at that position. The function can be coded using *Delete* and *InsertAfter* in the following manner:

```
var Replace = fun(A, B)
every a in A do
   Delete(a);
   InsertAfter(BeginTag(a), B)
end
```

Note that this encoding is only possible because the *Delete* function does not remove any tags from the page, and thus we can apply the *BeginTag* function to a deleted tag without causing an exception.

Examples:

```
// Make all links bold.
links = Elem(P, "a");
every L in links do
   Replace(L, NewPiece("<b>" + Markup(L) + "</b>"))
end

// Replace all links with the word "censored".
Replace(Elem(P, "a"), NewPiece("<i>censored<i>"))
```

TABLE 23. Page Modification Functions

Function	Description
Delete(s: pieceset): nil Delete(q: piece): nil	Deletes *s* or *q* from the page by removing all the pieces from the page data structure.
InsertBefore(t: tag, q: piece): nil InsertBefore(t: tag, s: pieceset): nil	Inserts a copy of *q* before the tag *t*. Inserts copies of the elements of s before the tag *t*.
InsertAfter(t: tag, q: piece): nil InsertAfter(t: tag, s: pieceset): nil	Inserts a copy of *q* after the tag *t*. Inserts copies of the elements of *s* after the tag *t*.
NewPiece(t1: tag, t2: tag): piece	Returns a new unnamed piece starting before *t1* and ending after *t2*.
NewPiece(q: piece): piece	Equivalent to *NewPiece(BeginTag(q), EndTag(q))*
NewNamedPiece(name: string, t1: tag, t2: tag): piece	Returns a new named piece starting before *t1* and ending after *t2*.
NewNamedPiece(name: string, q: piece): piece	Equivalent to *NewNamedPiece(name, BeginTag(q), EndTag(q))*.
Replace(a: pieceset, b: pieceset): nil	Replaces each piece set of *a* with copies of all the elements of *b*.

CHAPTER 5

Modules

WebL includes a number of standard modules for the convenient reuse of often required functionality. The purpose of this chapter is to introduce the more common modules shipped with the WebL installation (Table 24). To use most of these modules, a programmer must *import* the module and refer to the exported variables of the module. See “Modules” on page 46.

TABLE 24. Standard WebL Modules

Module	Function
Base64	Encodes and decodes base 64 strings used for user authentication at many web sites.
Browser	Provides access to the web browser for displaying web pages. (Page 116)
Cookies	Provides functionality to save and load the HTTP cookie database. (Page 117)
Farm	Introduces a technique for programming and controlling several concurrently executing threads. (Page 119)
Files	Functions to process local files and download pages to files. (Page 121)
Java	WebL-Java integration support. (Page 124)
Servlet	Java Servlet support. (Page 131)

TABLE 24. Standard WebL Modules

Module	Function
Str	General string related functions. (Page 136)
Url	Url manipulation functions. (Page 138)
WebCrawler	An extensible web crawler object. (Page 141)
WebServer	Implementation of a simple web server. (Page 143)

Module Base64

Base 64 encoding of strings is typically used to “scramble” transmitted passwords when accessing web pages that require user authentication. The typical pattern for basic HTTP authentication is as follows:

```
import Base64;

var A = "Basic " + Base64_Encode("user:pw");
var P = GetURL("http://...”,
    [..], [. Authorization = A .]);
```

In the code above *user* must be set to the user name and *pw* to the authentication password. The last object passed to the *GetURL* function contains the authentication header to send to the web server.

The *Base64* module is also used when authenticating users to Web proxies by adding a *Proxy-Authorization* header to the HTTP request:

```
import Base64;

var A = "Basic " + Base64_Encode("user:pw");
var P = GetURL(url, nil,
              [. "Proxy-Authorization" = A .]);
```

TABLE 25. Module Base64

Function	Description
Encode(s: string): string	Encodes a string in the base64 encoding.
Decode(s: string): string	Decodes a string in base64 encoding.

Module Browser

The *Browser* module provides a way to display markup in your web browser. On the Windows platform, the default installed browser will be started up to display the page. On UNIX platforms, WebL tries to communicate with an already running copy of *Netscape*. Note that to implement this functionality, WebL has to write the markup to a temporary file in the specified character encoding.

On the Windows platform only, module *Browser* also provides rudimentary support for inquiring and controlling a running copy of a Netscape browser with Dynamic Data Exchange (DDE). Specifically it is possible to detect what web page is being viewed in the browser, and to request Netscape to navigate to a specific URL. Both the support for viewing markup and the DDE functionality is bundled in a Windows platform-specific DLL called *weblwin32.dll*. The *readme.txt* file that is part of the WebL distribution contains instructions how to install this DLL on a Windows machine.

TABLE 26. Module Browser

Function	Description
GetCurrentPage(): object	(Windows only) Returns information about the currently viewed page in a running copy of Netscape. The object returned has string fields *url* and *title* that specifies the viewed URL and title of the viewed page respectively.
GotoURL(url: string): nil	(Windows only) Sends a request to a running copy of Netscape to navigate to this *url*.
ShowPage(s: string): nil	Displays the markup contained in *s* in a web browser (uses the default locale for externalizing the string).
ShowPage(s: string, charset: string): nil	As above, but ensuring that the string is externalized in the indicated character set. Values for charset might be “iso-8859-1”, “Unicode”, “UTF8”, etc.

Module Cookies

Module *Cookies* allows the programmer to perform some basic operations on the HTTP cookie database. The cookie database contain client-side state that web servers have requested WebL to store for them. By default the cookie database starts out empty with each WebL run, and fills up as cookies are set. At the end of the run, the cookie database is discarded. This is the default WebL behavior, and no programmer action is required.

The contents of the cookie database can be overriden by specifying a non-nil "Cookie" header field as part of the *GetURL* and *PostURL* functions. Furthermore, the *Save* and *Load* functions of the *Cookie* module can be used to save the database to a file, and later load it again. These functions are required if the cookie database is to transcend a single WebL session.

The external file format of the cookie database is a line per cookie, where each cookie is stored in the same format as received in the "Set-cookie" HTTP header. More details about the HTTP Set-cookie header can be found in the cookie specification from Netscape.

Mutiple Cookie Databases. By default, the cookie database is shared by all threads and web requests of the WebL process. However, it is sometimes useful to have groups of requests using logically separate cookie databases. WebL allows you to specify which cookie database to use for each web request (and if no database is specified, the default shared cookie database is used). Cookie databases have programmer defined names (strings), and are automatically allocated whenever they are first used. In particular, the "cookiedb" field of the options parameter to *GetURL* and *PostURL* functions, specify which cookie database is to be used for that requests. For example, the following web request reads (and also writes) cookies from and to a database called "DB1".

```
GetURL("http://www.abc.com", nil, nil,
       [. cookiedb="DB1" .])
```

The default cookie database is used when no "cookiedb" option is specified or the database name is the empty string. Note that, as explained before, all cookie databases are discarded after the WebL program ends i.e. the databases are not stored to disk. If you need this functionality, the *Load* and *Save* functions of the *Cookie* module allows you to read and write cookie databases from and to file storage.

TABLE 27. Module Cookies

Function	Description
Load(filename: string): nil	Adds the cookies in *filename* to the default cookie database.
Save(filename: string): nil	Saves the default cookie database to *filename*.
Load(filename: string, database: string): nil	Adds the cookies in *filename* to the cookie database named *database*.
Save(filename: string, database: string): nil	Saves the cookie database named *database* to *filename*.

Module Farm

Module *Farm* introduces the concept of a farm object (an object with a hidden implementation). A farm consists of a number of workers that process jobs. The *Perform* method of a farm object allows the programmer to insert a job into the job queue of the farm. Idle workers (those that are not doing something) periodically pick a job from the queue to perform. When the job queue is empty and no workers are working, we say that the farm is idle.

It is important to note that the workers are simple-minded in the sense that should an exception occur while performing a job, the job is terminated without any indication to the programmer, and the worker becomes idle again. It is thus advisable to include exception handling code in the job itself.

The *Perform* method of a farm uses a special calling convention. The argument of this method must be an expression denoting a function application. For example, say we would like to turn a function invocation of *F* with two arguments into a job, we must write:

```
var frm = Farm_NewFarm(10);
frm.Perform(F(a, b))
```

It is important to know that the arguments to the function application are evaluated before the job is started. A typical application of a farm object is the following stupid web crawler program with 10 parallel workers:

```
import Farm;

var F = Farm_NewFarm(10);
var ProcessPage = fun(url)
    var page = GetURL(url);
    every a in Elem(page, "a") do
        F.Perform(ProcessPage(a.href))
    end
end;

F.Perform(ProcessPage("http://www.nowhere.com"));
while !F.Idle() do
    Sleep(10000)
end
```

TABLE 28. Module Farm

Function	Description
NewFarm(nofworkers: int): object	Creates a farm object with the specified number of workers.

TABLE 29. Methods of Farm Objects

Method	Description
Perform(e): nil	Adds a task or job to the farm queue. An idle worker will eventually remove the job from the queue. Note that *e* must be an expression where a function is applied.
Idle(): bool	Returns true if the job queue is empty and all workers are idle.
Stop(): nil	Kills off all the jobs, and stops all of the workers. Adding jobs to the queue after this operation will have no effect.

Module Files

The *Files* module provides rudimentary functions for testing the existence of a file, saving and loading pages and strings to and from files, and downloading web content to a file. The *filename* argument to each of the functions in Table 30 is a filename in the syntax supported by the file system underlying WebL.

The combination of the *SaveToFile* and *Eval* functions allows some limited persistant storage for WebL. For example, the following program writes a set out to disk and reads it back in again to the variable *T*:

```
import Files;

var S = {1, 2, 4, 6};
Files_SaveToFile("test.tmp", ToString(S));
var T = Files_Eval("test.tmp")
```

Note that this technique can be used only for externalizing non-recursive values that do not contain functions or methods — the external format of those structures are not legal WebL programs.

TABLE 30. Module Files

Function	Description
AppendToFile(filename: string, val: string): nil	Appends *val* to the end of the file.
AppendToFile(filename: string, val: string, charset: string): nil	As above, but sets the character encoding to use. Typical encodings are "iso-8859-1", "UTF8", etc.
Exists(filename: string): bool	Determines if a file with the specified name exists.
LoadFromFile(filename: string, mimetype: string): page	Loads a page object from a file.
LoadStringFromFile(filename: string): string	Loads a file as a string object, using the default character encoding.

TABLE 30. Module Files

Function	Description
LoadStringFromFile(filename: string, charset: string): string	Loads a file as a string object, using the character encoding specified by *charset*. Typical values for charset are "UTF8", "Unicode", "iso-8859-1", etc.
SaveToFile(filename: string, val: string): nil	Saves the string val to the specified file.
SaveToFile(filename: string, val: string, charset: string): nil	As above, but overrides the default character encoding of the saved file. Typical encodings are “iso-8859-1”, “UTF8”, etc.
GetURL(url: string, filename: string): page GetURL(url: string, filename: string, param: {object,string}): page GetURL(url: string, filename: string, param: {object,string}, header: object): page GetURL(url: string, filename: string, param: {object,string}, header: object, options: object): page	Similar to the built-in *GetURL* function except that the retrieved document is saved to the indicated file. The *url*, *param*, *header* and *option* arguments are the same as the built-in *GetURL* function.
PostURL(url: string, filename: string): page PostURL(url: string, filename: string, param: {object,string}): page PostURL(url: string, filename: string, param: {object,string}, header: object): page PostURL(url: string, filename: string, param: {object,string}, header: object, options: object): page	Similar to the built-in *PostURL* function except that the retrieved document is saved to the indicated file. The *url*, *param*, *header* and *options* arguments are the same as the built-in *PostURL* function.
Eval(filename: string): any	The cotnets of *filename* is evaluated as a WebL program. The result returned is the value of the last statement in the program.
List(dirname: string): list	Returns the file names and directory names contained in directory *dirname*.

TABLE 30. Module Files

Function	Description
IsDir(name: string): bool	Checks whether *name* is a valid directory name or not.
IsFile(name: string): bool	Checks whether *name* is a valid file name or not.
Mkdir(name: string): bool	Attempts to create a new directory called *name*, and returns success or failure.
Delete(name: string): bool	Attempts to delete the file called *name*, and returns success or failure.
Size(name: string): int	Returns the size in bytes of the file called *name*.

Module Java

The *Java* module allows you to access Java classes, objects, and arrays directly from the WebL programming language. This functionality provides practically transparent access to any functionality provided by Java class library, at the extra run-time cost of translating between WebL and Java data types. The direction of access is purely from WebL to Java; transparent Java to WebL access is not possible without changes in the Java virtual machine. Note that using module *Java* requires a knowledge of Java itself and some knowledge about the WebL implementation, which is beyond the scope of this user manual.

The WebL-to-Java integration works by automatically "wrapping" Java objects, classes, and arrays with special WebL types, and performing transparent translation of WebL data types to Java data types and vice-versa. The Java module introduces two new WebL data types for this purpose. The WebL "j-object" type is a special object type that wraps Java objects and Java classes. The WebL "j-array" type wraps Java arrays.

Type j-object. Wrapping a Java object in a WebL j-object is transparent to the WebL programmer. From the WebL programmer's perspective, the object behaves exactly the same as a normal WebL object. That is, the fields and methods of a Java object is directly accessible from the WebL j-object. For example, the following WebL code creates a Java Date object, and calls some of its methods to print out some of the details of the data object:

```
import Java;

var D = Java_New("java.util.Date");

PrintLn("Today's date is ", D.toString());
PrintLn("Today is ", D.getMonth());
PrintLn(D);
```

Notice how the last line prints out the Java object itself. The console output from this statement might look as follows, which illustrates that the methods and fields of the Java date object are reflected 1-to-1 inside the WebL j-object:

```
[. "setYear" = < setYear(int): void>,
"getSeconds" = < getSeconds(): int>,
"parse" = < parse(java.lang.String): long>,
"setTime" = < setTime(long): void>,
"getDay" = < getDay(): int>,
```

```
"setHours" = < setHours(int): void>,
"setMonth" = < setMonth(int): void>,
"notifyAll" = < notifyAll(): void>,
"after" = < after(java.util.Date): boolean>,
"setDate" = < setDate(int): void>,
"getHours" = < getHours(): int>,
"setSeconds" = < setSeconds(int): void>,
"wait" = < wait(long): void
   wait(long,int): void wait(): void>,
"getMonth" = < getMonth(): int>,
"toString" = < toString(): java.lang.String>,
"UTC" = < UTC(int,int,int,int,int,int): long>,
"notify" = < notify(): void>,
"getYear" = < getYear(): int>,
"before" = < before(java.util.Date): boolean>,
"equals" = < equals(java.lang.Object): boolean>,
"getTime" = < getTime(): long>,
"getTimezoneOffset" = < getTimezoneOffset(): int>,
"getMinutes" = < getMinutes(): int>,
"hashCode" = < hashCode(): int>,
"getClass" = < getClass(): java.lang.Class>,
"getDate" = < getDate(): int>,
"setMinutes" = < setMinutes(int): void>,
"toGMTString" = < toGMTString(): java.lang.String>,
"toLocaleString" = < toLocaleString():
                                    java.lang.String>
.]
```

WebL-Java type conversion. Furthermore, automatic translation between WebL and Java data types is done when calling methods and constructors, or assigning values to object fields. Table 33, “Conversion of WebL types into Java types,” on page 130, shows with what Java types a specific WebL type is compatible with. Refer to this table when *calling* a Java method or constructor. Refer to Table 32, “Conversion of Java types into WebL types,” on page 129, to see how values returned from methods and field accesses are converted back into WebL types. Studying these two tables will show that the type conversion is mostly restricted to converting between primitive Java and WebL types. That means for example that WebL objects can only be passed to methods that accept the implementation type of WebL objects (*webl.lang.expr.ObjectExpr*). This is not restrictive as it might sound; many methods in the JDK accept *java.lang.Object*’s as arguments, which is of course a superclass of *webl.lang.expr.ObjectExpr*. For example, it becomes possible to insert WebL objects into Java hash tables.

Here is a more complicated example which reads and numbers the lines of a file called "test.txt":

```
import Java;

var System = Java_Class("java.lang.System");

var F = Java_New("java.io.File", "test.txt");
var R = Java_New("java.io.BufferedReader",
          Java_New("java.io.FileReader", F));

var c = 1;
var L = R.readLine();
while L != nil do
   System.out.print(c);
   System.out.println(" " + L);
   c = c + 1;
   L = R.readLine();
end;
```

After each occurrence of *R.readLine*, the resulting line is converted into a WebL string type. When *System.out.print(ln)* is called, the WebL types are converted back into the appropriate Java type.

Statics. The above example also illustrates how to access a static field (namely *System.out*): The *Java_Class* function wraps the class into a WebL object, from where the field can be accessed directly. In addition, the example also shows how to use constructors with arguments.

Overloading. WebL programmers should be aware that constructors and methods are often overloaded in Java. In this case, WebL will attempt to match the best constructor or method by comparing the actual arguments (provided in WebL) with the formal arguments of the constructors and methods in question. This might lead to problems when the matching involves numeric types. Suppose a Java method named *X* is overloaded three times, with single formal arguments of type *int*, *short*, and *byte* respectively. Which instance of *X* will be called when a formal argument of type *int* is used in a call the function? WebL's approach is to prefer the "widest" type, which in this case would be *X* with the formal of type *int*. Programmers should be aware that this simple heuristic might cause the "wrong" instance of the overloaded method to be invoked. There is no support for enforcing calls to a specific overloaded method in WebL.

Type j-array. In addition to the j-object type, the Java module also provides a new type called j-array that wraps Java arrays. The reasoning behind providing a separate array type instead of using an established data type such type list, is that Java arrays are fundamentally different from WebL data types. Java arrays are mutable (i.e. elements can be overwritten), whereas WebL types (except for type object) are immutable. Thus passing a WebL list to a method that expects a Java array begs the question want would happen if the method mutates the array.

The Java array support in module Java includes functions to allocate an array of a specific type and size (*Java_NewArray*), retrieve an element at a specific index (*Java_Get*), and overwritting an element at a specific index (*Java_Set*).

The following program allocates, writes and reads the elements of an array:

```
import Java;

var A = Java_NewArray("int", 10);
Java_Set(A, 0, 42);
PrintLn(Java_Get(A, 0));
Java_Set(A, 1, "hello");// -> Type mismatch exception
```

Java Classpath. The Java *CLASSPATH* environment variable must be set correctly to access the Java classes that are external to the classes in *WebL.jar*. Programmers should be aware that when using the *-jar* option of the Java runtime, classes are only searched for in *WebL.jar*. It is thus better to run WebL with the *-cp* Java runtime option, where the CLASSPATH must be specified explicitly.

TABLE 31. Module Java

Function	Description
New(classname: string, ...): j-object	Allocates a Java object using the specified class name, and optional constructor arguments. Valid classnames are Java primitive types ("int", "char", "short", etc.) or fully specified Java class names ("java.lang.String", "java.util.Vector", etc.).
Class(classname: string): j-object	Maps the specified class into a WebL object, allowing the static fields of the class to be accessed.
NewArray(classname: string, size: int): j-array	Allocates a Java array of the specified type and size.
Get(A: j-array, i: int): any	Retrieves index *i* of array *A*.
Set(A: j-array, i: int, v: any):any	Sets index *i* of array *A* to value *v*.
Length(A: j-array): int	Returns the length of the Java array.

TABLE 32. Conversion of Java types into WebL types

Java Type/class/value	Corresponding WebL type
null	nil
boolean	bool
char	char
java.lang.String	string
long	int
int	int
short	int
byte	int
float	real
double	real
Any array type	j-array
webl.lang.expr.ObjectExpr	object
webl.lang.expr.ListExpr	list
webl.lang.expr.SetExpr	set
webl.lang.expr.AbstractFunExpr	fun
webl.lang.expr.AbstractMethExpr	meth
webl.page.Page	page
webl.page.Piece	piece
webl.page.PieceSet	pieceset
webl.page.TagExpr	tag
Any Java array	j-array
Any other class no listed above	j-object

TABLE 33. Conversion of WebL types into Java types

WebL Type	Compatible Java type/class/value
nil	null
bool	boolean
char	char, string
string	java.lang.String (and superclasses)
int	int, long, short, byte, float, double
real	float, double
object	webl.lang.expr.ObjectExpr (and superclasses)
list	webl.lang.expr.ListExpr (and superclasses)
set	webl.lang.expr.SetExpr (and superclasses)
fun	webl.lang.expr.AbstractFunExpr (and superclasses)
meth	webl.lang.expr.AbstractMethExpr (and superclasses)
page	webl.page.Page (and superclasses)
piece	webl.page.Piece (and superclasses)
pieceset	webl.page.PieceSet (and superclasses)
tag	webl.page.TagExpr (and superclasses)
j-object	Corresponding type of wrapped Java object.
j-array	Corresponding type of wrapped Java array type.

Module Servlet

Many web servers today support the *Java Servlet* standard from JavaSoft. This standard allows the efficient execution of server-side actions. In addition to the built-in Web server support (see module *WebServer*), WebL also supports the servlet standard directly and transparently. In fact, the WebL Servlet integration is so transparent that no new functions need to be introduced. The description how to use servlets provided below assumes a fair knowledge about Java and servlets; it is thus advisable to study the servlet documention before continuing.

Servlet access. The class *weblx.servlet.Servlet* implements the WebL servlet. By placing this class (or the jar file it is in, namely *WebL.jar*) on a servlet enabled webserver, it becomes possible to execute WebL code directly on the server. Web surfers may access your WebL servlet by accessing a URL of (typically) the following form:

```
http://www.host.com/servlet/weblx.servlet.Servlet/
modulename_variablename?arguments
```

In case your web server supports aliases, you can alias "weblx.servlet.Servlet" as "webl", which allows access from the following URL:

```
http://www.host.com/servlet/webl/
modulename_variablename?arguments
```

In both cases, *modulename* identifies the WebL module that contains the WebL servlet script, and *variablename* identifies an exported variable in that module. The value type of this variable must be a function with two formal arguments. The module will be loaded automatically the first time the URL is accessed (this happens only once; afterwards the module is cached).

Table 34 and Table 35 show the format of the two arguments of the function. The first is the request object, and the second is the response object. The explanation for the field names and values is found in the Java servlet specification available from Javasoft.

You may modify your WebL servlets while being used. WebL checks before each servlet access whether the WebL module has changed or not (using the file last-modified date). If the modified date is different from the modified date when the module was loaded first, the module is auotmatically reloaded.

Examples. The following WebL servlet shows how to set and retrieve a variable on your Web server.

```
// File: Example1.webl

var theval = nil;       // the variable

// To set the variable to "hello",  access:
//   http://www.host.com/servlet
//     /webl/Example1_SetVal?x=hello

export var SetVal = fun(req, res)
   theval = req.param.x;
   res.mimetype = "text/plain";
   res.result = "Set val to " + theval;
end;

// To retrieve the value, access:
//   http://www.host.com/servlet/webl/Example1_GetVal

export var GetVal = fun(req, res)
   res.mimetype = "text/plain";
   res.result = "Val is " + theval;
end;
```

Note how the *x* parameter in the URL is accessed with *req.param.x*. Note, that in the case of multiple parameters with the same name, the particular parameter field will have a list of strings as value. Programmers should thus be aware of the fact that the value type of parameters is either a string, or a list of strings, depending on the number of parameters.

Often during servlet development you will need to snoop the servlet request headers. The following example shows how this is done with a WebL servlet:

```
// File: Example2.webl

var Decode = fun(req)
   var s = "Header snoop:\n\n";
   every field in req do
      s = s + ToString(field) + ": "
            + ToString(req[field]) + "\n";
   end;
   s;
end;
```

```
// Access the following URL:
//   http://www.host.com/servlet/webl/Example2_Snoop
//
export var Snoop = fun(req, res)
   res.mimetype = "text/plain";
   res.result = Decode(req);
end;
```

Servlets typically use HTTP cookies to keep track of client state. The following WebL servlet maintains a visit counter inside the client's cookie:

```
// File: Example3.webl

// Access the counter with:
//   http://www.host.com/servlet/webl/Example3_Count

export var Count = fun(req, res)
   res.result = "Cookie test\n";

   // Retrieve the cookie named "cc"
   var count = ToInt(req.cookies.cc) ?
      begin     // executed if no such cookie exists
         res.result = res.result + "No cookie\n";
         0
      end;

   res.result = res.result + "Count = " + count;
   res.mimetype = "text/plain";

   // set the new cookie
   res.cookies = [.
      cc = [. domain="www.myhost.com", path="/",
             value=count + 1, comment="",
             maxage=-1, version=0 .]
    .]
end;
```

Server setup. Servlet setup can be complicated. First make sure that you can access the demo servlets that come with your web server. Only then continue with this checklist:

1. Put *WebL.jar* is the CLASSPATH of your web server (server dependent).
2. Add a *configuration parameter* for the WebL servlet to your Web server (server dependent):
   Parameter name: *webl.path*
   Parameter value: (directory search path for WebL scripts)
3. Restart your web server for changes to take affect.
4. Place WebL scripts in the directory of the search path, as indicated by **2**.

TABLE 34. Format of the Servlet request parameter object

```
[.
method: string,
requestURI: string,
servletpath: string,
pathinfo: string,
pathtranslated: string,
querystring: string,
remoteuser: string,
authtype: string,

remoteaddr: string,
remotehost: string,
scheme: string,
servername: string,
serverport: string,
protocol: string,
contenttype: string,

header: object,
param: object,
cookies: object]
.]
```

TABLE 35. Format of the Servlet response parameter object

```
[.
statuscode: int,
statusmsg: string,
result: string,
mimetype: string
header: [. name=val, name=val, ... .],
cookies : [.
  cookiename = [.
      comment: string,
      domain: string,
      maxage: int,
      path: string,
      secure: bool,
      value: string,
      version: int
    .],
  cookiename = ...,
  cookiename = ...,
  ...
  .]
.]
```

Module Str

The *Str* module provides several useful operations on string values.

TABLE 36. Module Str

Function	Description
Compare(a: string, b: string): int	Returns -1, 0, +1 if *a* is less than, equal, or greater than *b*.
EndsWith(s: string, regexp: string): bool	Tests if a string ends with a particular pattern.
EqualsIgnoreCase(a: string, b: string): bool	Tests if *a* and *b* are equal in a case-insensitive manner.
IndexOf(pat: string, s: string): int	Returns the first position where *pat* occurs in s, otherwise -1.
LastIndexOf(pat: string, s: string): int	Returns the last position where *pat* occurs in s, otherwise -1.
Match(s: string, regexp: string): object	Tests whether *s* matches the regular expression *regexp*. If so, an object is returned where the fields of the object are integers numbered from 1 onwards, each of them having a value corresponding to the Perl5 groups (as indicated by the parenthesis sub-expressions in *regexp*) that has been matched. *Nil* is returned otherwise.
Replace(s: string, from: char, to: char): string	Replaces each occurrence of *from* with *char* in *s*.
Search(s: string, regexp: string): list	Searches for all the occurrences of the regular expression *regexp* in *s*, and returns a list of objects for each of them. The fields of the objects are similar to those returned by the *Match* function. Note that object field 0 is the complete matched character string.

TABLE 36. Module Str

Function	Description
Split(s: string, chars: string): list	Splits the string s at positions where any of the characters of *chars* appear. The function returns a list of strings.
StartsWith(s: string, regexp: string): bool	Tests if a string starts with a particular pattern.
ToLowerCase(s: string): string	Turns *s* into lowercase.
ToUpperCase(s: string): string	Turns *s* into uppercase.
Trim(s: string): string	Remove white space characters like new lines, carriage returns, tabs, etc. from the beginning and end of the argument string.

Module Url

The *Url* module performs a number of useful operations on URL strings. For example, it is sometimes useful to break up a URL into its constituent parts, modify some of them, and glue the parts back together again. In the same manner, query strings (the part of a URL that follows the question mark) also need to be manipulated.

For example, given the URL (a typical AltaVista query)

```
http://www.altavista.digital.com/cgi-bin/query
?pg=q&kl=XX&q=%2Bjava+%2Bcoffee
```

the *Split* function will return an object as follows:

```
[.
   query = "?pg=q&kl=XX&q=%2Bjava+%2Bcoffee",
   path = "/cgi-bin/query",
   host = "www.altavista.digital.com",
   ref = "",
   scheme = "http"
.]
```

Applying the *SplitQuery* function to the *query* field of this object will return the following object:

```
[. kl = "XX", pg = "q", q = "+java +coffee" .]
```

Behind the scenes, decoding the query string involves calls to the *Decode* function to remove the character encodings (eg. %@B and so on).

The *Glue* and *GlueQuery* functions will glue those objects back together again. The field names generated by the *Split* function are summarized in Table 38. Note that the current implementation can process only the *http*, *ftp*, and *file* protocol schemes.

TABLE 37. Module Url

Function	Description
Decode(s: string): string	Decodes a string in the MIME type encoding "x-www-form-urlencoded". The complementary function is called *Encode*.
Encode(s: string): string	Encodes a string in the MIME type encoding "x-www-form-urlencoded" (which is generally used to encode input form parameters). The complementary function is called *Decode*.
Glue(obj: object): string	Takes the constituent parts of a URL (as broken up by *Split*), and glues them together to form a URL again.
GlueQuery(obj: object): string	Given an object constructed from *SplitQuery*, *GlueQuery* returns the original query string again.
Resolve(base: string, rel: string): string	Given the base URL *base* and the relative URL *rel*, return the resolved URL.
Split(url: string): object	Splits a URL into its constituent parts (like *scheme*, *host*, *path*, *query* etc.). Each part becomes a field of the object returned.
SplitQuery(query: string): object	Splits a *query* string into its constituent parts. Each part becomes a field of the object returned. Query strings typically follow the ? in URLs.

TABLE 38. URL constituents

Field name	Used in schemes	Description
scheme	All	http, ftp, file, etc.
host	http, ftp, file	Host name.
port	http, ftp	TCP/IP connection port.
path	http, ftp, file	Full path name of the addressed resource.
query	http	Query string (after '?' in URL)
ref	http, file	Anchor reference string (after '#' in URL).
user	ftp	Login user name.
password	ftp	Login password.
type	ftp	File transfer type 'a', 'i', 'd'.
url	Unknown schemes	Contains the complete URL because no constituents could be extracted (because the scheme is unknown).

Module WebCrawler

Module *WebCrawler* exports a single object called *Crawler* that implements a low-performance multi-threaded web crawler. To use the web crawler, the methods *Visit* and *ShouldVisit* must be overridden by the programmer. The *Visit* method is called by the crawler each time a page is visited, and the *ShouldVisit* method returns true when a specific URL must be crawled. The crawler is activated by the *Start* method which takes as argument an integer specifying how many threads should perform the crawl. At this point the crawler has no pages to crawl yet. Pages are inserted into the crawler queue with the *Enqueue* method. As each page in the queue is processed, the crawler will extract all the anchor ("<A>") tags in that page, and call the *ShouldVisit* method to determine if the page referred to by the anchor should be crawled or not. The *Abort()* method can be called at any time to terminate the crawl.

The following example implements a web crawler that prints the URL and title of all pages visited. The crawl is restricted to pages on the *pa.dec.com* domain that have a URL that ends in a "/", ".htm" or ".html". The queue is initially seeded with two pages from where the crawl will progress in a breadth-first fashion. Note that the program finally goes to sleep while the crawl is in progress.

```
import Str, WebCrawler;

var MyCrawler = Clone(WebCrawler_Crawler,
    [.
   Visit = meth(s, page)
      var title = Text(Elem(page, "title")[0]) ?
                    "This page has no title";
      PrintLn(page.URL, " title=", title)
   end,

   ShouldVisit = meth(s, url)
      Str_StartsWith(url,
          `http://www-\w*[.]pa[.]dec[.]com`)
      and
      Str_EndsWith(url, "(/)|([.]html?)")
   end,
   .]);
MyCrawler.Start(2);// Only two threads are used.
MyCrawler.Enqueue("http://www-src.pa.dec.com/");
MyCrawler.Enqueue("http://www-wrl.pa.dec.com/");
```

```
Stall();
```

Module WebServer

The *WebServer* module exports an interface to a simple multi-threaded web server. The *Start* function allows the programmer to start the web server on a specific port on the host machine where WebL is running. After starting the web server, HTML and other files will be served from the *fileroot* directory indicated when the server was started. The programmer may publish WebL *functions* to be executed when specific URL paths are requested.

For example, the following program starts the web server on port 90, and publishes a function called *Echo* that returns an HTML page. Afterwards, its goes to sleep while requests are serviced. If a request for the URL /bin/echo is received by the server, the *Echo* function is invoked.

```
import
    WebServer;

WebServer_Start("c:\\InetPub\\wwwroot", 90);

var Echo = fun(req, res)
  res.result = "<html><body>Hello!</body></html>";
end;

WebServer_Publish("/bin/echo", Echo);

while true do
    Sleep(10000)
end
```

Functions may be *published* under any (case-sensitive) name. The web server will first consult the list of exported functions when a request is received, by comparing the path of the URL requested to each of the given names of the published functions. Should no published name match the URL path requested, the web server attempts to serve a file in the directory rooted by *fileroot.*

The formal arguments *req* and *res* represent respectively the request the web server received and the response the web server has to return. The idea is that the invoked function looks at the object *req* to figure out what to do, and modifies the object *res* to tell the server what to do (i.e. what data to return, etc.). For example, given the following request to the web server (running on a machine called *ck.pa.dec.com*):

```
http://ck.pa.dec.com:90/bin/echo?x=3&y=abc+def
```

the *Echo* function could be invoked with the following *req* object:

```
[.
   contents = "",
   protocol = "HTTP/1.0",
   method = "GET",
   uri = "/bin/echo?x=3&y=abc+def",
   path = "/bin/echo",
   query = "x=3&y=abc+def",
   param = [. y ="abc def", x = "3" .],
   header = [.
      "Accept-Charset" = "iso-8859-1,*,utf-8",
      Connection = "Keep-Alive",
      "User-Agent" = "Mozilla/4.04[en] (WinNT; I)",
      Accept = "image/gif, image/x-xbitmap,
         image/jpeg, image/pjpeg, image/png, */*",
      "Accept-Language" = "en",
      Host = "ck.pa.dec.com:90"
   .]
.]
```

and the following *res* object:

```
[.
   result = nil,
   header = [.
      Server = "WebL",
      Date = "Thu May 14 15:59:01 PDT 1998",
```

```
            Content-Type" = "text/html"
        .],
        statuscode = 200,
        statusmsg = "OK"
    .]
```

The invoked function can now look at the fields of *req* to determine how to handle the request, and modify the fields of *res* to indicate the result to be returned (note that many of the fields are filled in to sensible values when the function is invoked). The meaning of the individual fields of the request and response object are listed in Table 40 and Table 41 respectively (note that all fields except *statuscode* are of value type string or object). The most commonly used field is *param*, which indicates the request parameters received.

TABLE 39. Module WebServer

Function	Description
Start(fileroot: string, port: int): nil	Starts the web server on the indicated port, and prepares to server up files located at fileroot in the file system.
Stop(): nil	Stops the web server.
Publish(name: string, f: fun): nil	Publishes the function *f* under a *name* on the server. The name indicates the URL that will invoke function *f*. Function f has to be a function with two formal arguments (see discussion above).

TABLE 40. Fields of the Request Object

Field	Description
method	HTTP method GET, POST, etc.
protocol	The HTTP protocol version.
uri	The URL of the complete request.
query	The query part of the request.
path	The path of the script requested.
contents	The contents of the request message (typically only has a value for POST methods).
param	Object with fields submitted in either a GET or POST method. In case a particular parameter is repeated in the request, the appropriate field of the the *param* object will be set to a list of strings, corresponding to the individual parameter values.
header	Header fields the browser sent with the HTTP request. In case a particular header field is repeated in the request, the appropriate field of the the *header* object will be set to a list of strings, corresponding to the individual header field values.

TABLE 41. Fields of the Response Object

Field	Description
statuscode	Integer status code to be returned. The semantics of the codes are listed in the HTTP specification.
statusmsg	Status message that matches this status code.
result	The page that is to be returned to the client.
header	The header fields the server will return to the client.

CHAPTER 6

Examples

The purpose of this chapter is to give a feeling for how WebL can be used in real-world programs. It contains three case studies:

- Calculating statistics on newspaper articles,
- The implementation of a simple multi-threaded web crawler class,
- The implementation of a highlight proxy.

Reading Grades

The following program calculates the *Kincaid* score of a set of headline newspaper articles found on the *www.news.com* web server, and outputs a sorted table of those article titles to the file “kincaid.txt”. The Kincaid scoring function is used to judge reading ease of an English document based on its sentence and word characteristics. The function’s output ranges from 5.5 to 16.5 in reading grade level.

Note that this implementation does not calculate the correct Kincaid reading grade as it takes some shortcuts in calculating the number of sentences and syllables in a page. Also, web pages tend to contain a lot of headings and so on, which are not identified correctly as sentences. Web pages differ enough from the Navy manuals

(on which the scoring function is based) to let us conclude that we are only calculating a relative score between similar pages in a corpus.

```
1   import Str, Files;
2
3   var Scores = fun(page)
4      var txt = Text(page);
5      var letters = Size(Str_Search(txt, "[A-Za-z]"));
6      var words = Size(Str_Search(txt, "[0-9a-zA-Z']+"));
7      var syllables = Size(Str_Search(txt, "[aeiouy]+"));
8
9      var exceptions = Pat(page, "[ ]([A-Z0-9][.])+");
10     Replace(exceptions, NewPiece(" X ", "text/plain"));
11     var sentences = Size(Pat(page, "[.]"));
12
13     [.
14        Sentences = sentences,
15        Words = words,
16        Syllables = syllables,
17
18        ARI = 4.71 * (letters / words) +
19                  0.5 * (words /sentences) - 21.43,
20        Kincaid = 11.8 * (syllables / words) +
21                  0.39 * (words / sentences) - 15.59,
22        CLF = 5.89 * (letters / words) -
23                  0.3 * (sentences / (words / 100)) - 15.8,
24        Flesch = 206.835 - 84.6 * (syllables / words)
25                  - 1.015 * (words / sentences)
26     .]
27  end;
28
29  var ScorePageList = fun(L)
30     var res = [];
31     var count = 1;
32     every s in L do
33        try
34           PrintLn(count, " scoring ", s);
35           count = count + 1;
36           var page = GetURL(s);
37           var sc = Scores(page);
38           sc.URL := s;
39           sc.Title := Text(Elem(page, "title")[0]);
40           res = res + [sc];
41        catch e          // just report errors
42           on true do PrintLn(e.msg)
```

```
43          end;
44      end;
45      res
46  end;
47
48  var FollowLink = fun(page, anchortext)
49      var dest = (Elem(page, "a") contain
50                                  Pat(page, anchortext))[0];
51      GetURL(dest.href)
52  end;
53
54  var GetStories = fun()
55      var res = [];
56      var P = GetURL("http://www.news.com");
57      var H = FollowLink(P, "(?i)all the headlines");
58      var A = (Elem(H, "a") directlyafter Pat(H, "&#149"))
59                          !inside Elem(H, "strong");
60      every a in A do
61          res = res + [a.href];
62      end;
63      PrintLn(Size(res), " articles found.");
64      res
65  end;
66
67  var pages = GetStories();
68  var res = ScorePageList(pages);
69
70  res = Sort(res,
71      fun(a, b)
72          var diff = a.Kincaid - b.Kincaid;
73          if diff > 0.0 then 1
74          elsif diff == 0.0 then 0
75          else -1
76          end
77      end);
78
79  var s = "";
80  every x in res do
81      PrintLn(x.Kincaid, " ", x.Title);
82      s = s + x.Kincaid + " " + x.Title + "\r\n";
83  end;
84  Files_SaveToFile("kincaid.txt", s);
```

Lines 3-24 implement the core of the scoring function. After extracting the text of the page (line 4), we proceed to calculate the number of letters (line 5), words (line 6), and syllables (line 7) on the page, using a few simple regular expressions. Lines 9-11 take care of removing any initials that might appear in the page. This is necessary as the number of periods in the page is used as a measure of how many sentences are present, and initials containing periods will skew that count. Finally lines 13-26 calculate a few common reading scores, and return a "score" object with the results to the caller.

Lines 29-46 calculate reading scores for a list of URLs. Lines 36-37 fetch the individual pages and calculate the score. In lines 38-39 we extend the score object with fields to identify the URL and title of the page. The *ScorePageList* function returns a list of these score objects.

The purpose of the *GetStories* function (lines 54-65) is to retrieve a list of URLs representing newspaper articles. After fetching the root page from *news.com* (line 56), we follow the link called "All the headlines" to a page that contains all the stories of the day (line 57). Lines 58-59 perform the extraction of the story URLs. We locate all the anchors appearing after a bullet symbol (identified by the "&#149"; character entity) that are not written in the *strong* font. Lines 60-62 construct a list object of all the URLs found.

The main program starts at line 67. First we fetch the stories, and then score them. Lines 70-77 take care of sorting the stories according to score.

Finally, lines 79-84 take care of printing the result and writing it to a file.

WebCrawler

In this example, we illustrate how to build a simple web crawler framework that can easily be customized. The basic idea is to define a generic *Crawler* object of which methods can be overridden to customize its behavior. By the way, our crawler implementation is provided as standard in WebL in a module called *WebCrawler.*

First we define the generic *Crawler* object as follows:

```
1   import Str, Farm;
2
3   export var Crawler =
4       [.
5           // Pages visited so far (and removed from queue)
6           // and pages waiting in the queue.
7           enqueued = [. .],
8
9           // Will contain the farm after the start method is
10          // called.
11          farm = nil,
12
13          // Method that should be overridden.
14          Visit = meth(s, page) PrintLn(page.URL) end,
15          ShouldVisit = meth(s, url) true end,
16
17          Enqueue = meth(s, url)
18              // First remove everything following #
19              var pos = Str_IndexOf("#", url);
20              if pos != -1 then
21                  url = Select(url, 0, pos)
22              end;
23              lock s do
24                  var present = s.enqueued[url] ? false;
25                  if !present and s.ShouldVisit(url) then
26                      s.enqueued[url] := true;
27                      s.farm.Perform(s.ProcessPage(s, url))
28                  end
29           end
30      end,
31
32      ProcessPage = fun(s, url)
33          try
34              var page = GetURL(url); // fetch the page
```

```
35              s.Visit(page);
36
37              // Process all the links from this page.
38              every a in Elem(page, "a") do
39                  s.Enqueue(a.href) ? nil
40              end
41          catch E
42              on true do PrintLn(url, " err: ", E.msg)
43          end;
44      end,
45
46      Start = meth(s, noworkers)
47          s.farm = Farm_NewFarm(noworkers);
48      end,
49
50      Abort = meth(s) s.Stop() end
51  .];
```

First we need to keep track of all pages visited so far with an associative array (aka a WebL object) where the fields are the visited URLs, and the value is either true or false (line 7). Note that an alternative implementation could use a set instead of an object without any performance penalty.

Lines 14 and 15 define the two methods that need to be overridden to customize the crawler. The *Visit* method is called each time a new page is visited, and the *Should-Visit* method indicates whether a specific URL should be crawled or not.

The *Enqueue* method (lines 17-30) adds a URL to the queue of pages to be fetched. The first task is to strip off any references from the URL (lines 19-22). Line 24 then checks if we visited the page already. Note the use of the ? service combinator to catch the exception should the URL not be in the visited array. If the URL is not present, and we should visit this page (line 25), we remember that we have seen the page (line 26), and then pass the job of retrieving the page to a farm object (line 27).

Eventually when a worker on the farm reaches a new job, the *ProcessPage* function is invoked (lines 32-44). After the page is fetched (line 34), we call the method *Visit* to let the crawler process the page (line 35). Lines 38-40 take care of enqueing all the anchors found on the page.

A custom crawler. Now we look at how we can create a custom crawler using the generic crawler above. To override the *Visit* and *ShouldVisit* methods, we use the

Clone builtin applied to the generic crawler and our own object that contains the modifications to the generic crawler we would like to make (lines 3-16).

```
1   import Str, WebCrawler;
2
3   var MyCrawler = Clone(WebCrawler_Crawler,
4       [.
5           Visit = meth(s, page)
6               var title = Text(Elem(page, "title")[0]) ? "N/A";
7               PrintLn(page.URL, " title=", title);
8           end,
9
10          ShouldVisit = meth(s, url)
11              Str_StartsWith(url,
12                  `http://www-\w*\.pa\.dec\.com`)
13              and
14              Str_EndsWith(url, "(/)|(.html?)")
15          end,
16      .]);
17
18  MyCrawler.Start(2);
19  MyCrawler.Enqueue("http://www-src.pa.dec.com/");
20  MyCrawler.Enqueue("http://www-wrl.pa.dec.com/");
21  while !MyCrawler.Idle() do Sleep(10000) end
```

Our particular implementation of the *Visit* method extracts and prints the URL and title of the page (lines 5-8). The *ShouldVisit* method (lines 10-15) restricts crawling to host names of the form "*www-*.pa.dec.com"* and URLs that end either in "/" or ".html".

Lines 18-20 start up the crawler with two workers and enqueue two starting point URLs. Line 21 goes in a loop that checks every 10 seconds whether the workers have become idle, in which case the crawler terminates.

Highlight Proxy

In this example we illustrate how to perform transformations on viewed pages in a proxy like fashion. In particular, we would like to build a highlight proxy that highlights all occurences of a particular word on the Web in red. The highlight proxy is contacted with

```
http://www.host.com:9092/bin/highlight?url=X&word=Y
```

where *X* denotes the starting point URL on the Web, and *Y* denotes the word that is to be highlighted (and of course *www.host.com* is the machine the proxy server is run on). Our proxy is written in such a way that all links that are followed from page X onwards, are redirected to our proxy again. This is accomplished by rewritting the contents of the page.

```
1   import Url, WebServer;
2
3   var port = 9092;
4   var where = "/bin/highlight";
5
6   var Highlight = fun(req,res)
7     var url = req.param.url ? "http://www.compaq.com";
8     var word = req.param.word ? "Compaq";
9     var page = GetURL(url);              // fetch the page
10
11    every w in Pat(page,word) !inside Elem(page,"title") do
12      // wrap a font element around it
13       var p = NewNamedPiece("font",w);
14       p.size := "+1";           // define its size attribute
15       p.color := "red";         // define its color attribute
16    end;
17
18    every a in Elem(page, "a") do
19      a.href = where +
20            "?word=" + Url_Encode(word) +    // word parameter
21             "&url=" + Url_Encode(a.href)     // url parameter
22        ? nil;                           // nothing if no href
23    end;
24    res.result = Markup(page);           // this is the result
25  end;
26
27  WebServer_Publish(where, Highlight);
28  WebServer_Start("/dev/null",port);
```

```
29  Stall()
30
```

The highlight proxy consists of single function called *Highlight* (lines 6-25). This function is exported with the built-in WebL web server in lines 27 and 28. (More information about the built-in Web server can be found in the *WebServer* module documentation.) On lines 7 and 8 we extract the URL and word parameters passed to the proxy. Note how we use service combinators to provide sensible defaults in case no parameters are present.

Lines 11-16 does the actual highlighting of the word on the page. In line 13 a "font" element is wrapped around the occurrence of the word. In addition, lines 14 and 15 define the size and color attributes of the new font element. Note that we also make sure in line 11 that we only wrap word occurences outside of the title of the Web page.

The next step is to rewrite the *href* attribute of all anchors ("a" elements) in the page to work correctly with our proxy. This involves passing the old *href* attribute as the URL parameter to our proxy. We use the *Url_Encode* function to encode special characters in the URL as dictated by the URL specification. Finally, in line 24 we re-generate the markup of the (now modified) page, and return it back to the browser by assigning it to the appropriate field of the server response object *res*.

CHAPTER 7

WebL Quick Reference

This chapter is a quick reference to the WebL programming language. It contains the WebL EBNF syntax, operator precedence table, list of operators and functions, and the Perl5 regular expression format specification.

Running WebL Programs

Running a WebL program is highly dependent on the host platform. The WebL classes and resources are bundled in a Java JAR file called *WebL.jar*. The main class in this JAR file is called *WebL.class*. The main method of this class needs to be executed with the following arguments:

```
{options} filename [arg1 arg2 ... ]
```

The options are summarized in Table 42. The *filename* argument specifies the name of WebL program to be executed, and *arg1*, *arg2*, etc. are the arguments passed to the program. The latter argument list can be accessed from the variable called *ARGS* inside WebL programs.

The following list gives an indication of how WebL programs can be executed depending on one of several Java installation scenarios:

```
Java development kit:
   java WebL {options} filename [arg1 arg2 ...]

Java Runtime Environment:
   jre -cp WebL.jar WebL {options} filename
                            [arg1 arg2 ...]

Java 2 (a.k.a. JDK 1.2) with extension support:
   java -jar WebL.jar {options} filename
                            [arg1 arg2 ...]
```

TABLE 42. WebL Command Line Options

Option	Description
-D	Emit casual debugging output.
-Llogfile	Write casual debugging output to a log file.
-C	Print performance counters at end of run.
-P	Wait for ENTER when the program finishes.

Script search path. By default WebL will search for scripts and modules in the current working directory and in the */scripts* sub-directory inside the *WebL.jar* file. The directory search path can be changed by setting a Java system property called "webl.path" to a set of directories. This can be done on the command line with the "D" option:

```
java -Dwebl.path=dir1;dir2;dir3 WebL ...   (Windows)

java -Dwebl.path=dir1:dir2:dir3 WebL ...   (Unix)
```

Note that setting a "webl.path" shell environment variable won't do because environment variables are not accessible from Java applications.

Java System Properties. WebL programmers can access the system properties of the underlying Java implementation through a global WebL object called *PROPS*. For example, to access the user name of the person executing the script, you can write:

```
PROPS["user.name"]
```

The following *PROPS* object gives an idea of what information is accessible from here:

```
[.
    "path.separator" = ";",
    "ftpNonProxyHosts" = "*.pa.dec.com",
    "http.proxyHost" = "www-proxy1.pa.dec.com",
    "http.nonProxyHosts" = "*.pa.dec.com",
    "http.proxyPort" = "8080",
    "user.language" = "en",
    "ftpProxyHost" = "www-proxy.pa.dec.com",
    "user.region" = "US",
    "ftpProxyPort" = "8080",
    "java.vendor" = "Sun Microsystems Inc.",
    "file.encoding" = "8859_1",
    "line.separator" = "\n",
    "file.encoding.pkg" = "sun.io",
    "os.name" = "Windws NT",
    "user.name" = "marais",
    "awt.toolkit" = "sun.awt.windows.WToolkit",
    "java.class.version" = "45.3",
    "file.separator" = "\",
    "http.proxySet" = "true",
    "user.timezone" = "PST",
    "java.home" = "C:\JAVA",
    "java.version" = "11",
    "os.arch" = "x86",
    "java.vendor.url" = "http://www.sun.com/",
    "ftpProxySet" = "false",
    "os.version" = "4.0",
    "user.dir" = "C:\Proj\WebL3.0\java",
    "user.home" = "Z:\marais",
    "java.class.path" = "."
.]
```

WebL EBNF

WebL programs can be written in the Unicode character set (little or big-endian byte ordering with an initial Unicode byte ordering mark) or the more compact UTF-8 character set. Note that the first 127 characters of UTF-8 correspond to the widely used western ISO-8859-1 or Latin 1 character set.

White space and comments are ignored in WebL programs. Comments consist of either:

- a double forward slash token //, which introduces a comment till the end of the line, or
- the token pairs /* and */ with comments in between.

Note that comments of the style /* */ may nest.

The WebL EBNF is:

```
Module      = { Import } SS
Import      = import [ Ident { "," Ident } ] ";"
SS          = (Var | E) { ";" (Var | E) } [ ";" ]
Var         = [ export ] var IdentInit { "," IdentInit }
IdentInit   = Ident [ "=" E ]
E           =   Value
            | ImportRef
            | E BinOp E
            | UnOp E
            | Statement
            | "(" E ")"
            | E "(" [ E { "," E } ] ")"
            | FieldRef
            | FieldRef ":=" E              // define expr
Value       = nil | Bool | String | Real | Integer | Character | Object | Set | List
ImportRef   = Ident [ "_" Ident ]
Bool        = true | false
Object      = "[." [ Field { "," Field } ] ".]"
Field       = Ident "=" E
Set         = "{" [ E { "," E } ] "}"
List        =   "[" [ E { "," E } ] "]"
            | "[|" [ E { "," E } ] "|]"
FieldRef    =   E "[" E "]"
            | E "." Ident
BinOp       =   "+" | "-" | "*" | "/" | div | mod
            | "<" | "<=" | "==" | "!=" | ">" | ">="
            | and | or
            | "|" | "?"
            | "="
            | member
            | inside | !inside | directlyinside | !directlyinside
            | contain | !contain
            | directlycontain | !directlycontain
            | after | !after | directlyafter | !directlyafter
            | before | !before | !directlybefore | directlybefore
            | overlap | !overlap
            | without
UnOp        = "-" | "+" | "!"
Statement   = WhileStat | IfStat | FunStat | MethStat | CatchStat
            | EveryStat | LockStat | RepeatStat | BeginStat | ReturnStat
WhileStat   = while SS do SS end
IfStat      = if SS then SS [ ElseStat ] end
ElseStat    = else SS | elsif SS then SS [ ElseStat ]
```

```
FunStat      = fun "(" [ Ident ( "," Ident } ] ")" SS end
MethStat     = meth "(" [ Ident ( "," Ident } ] ")" SS end
CatchStat    = try SS catch Ident { on E do SS } end
// Ident introduced into a new scope
EveryStat    = every Ident in E do SS end
// Ident introduced into a new scope
LockStat     = lock SS do SS end
RepeatStat   = repeat SS until SS end
BeginStat    = begin SS end
ReturnStat   = return [E]

Ident        = Letter { Letter | Digit }
Integer      = Digit { Digit }
Real         = Integer [ Fraction ] [ Exponent ]
Fraction     = "." Integer
Exponent     = ( "e" | "E" ) [ "+" | "-"] Integer
String       = "\"" { Char } "\"" | "‘" { Char } "‘"
Char         = "’" Char "’"
Digit        = "0" .. "9"
Letter       = "a" .. "z" | "A" .. "Z"
Char         =  *any unicode character*
```

Strings and characters may contain the escapes listed in Table 43. To write the non-standard escapes that occur in regular expressions (like \w and \d), it is advisable to use back-quoted strings which ignore the string content completely.

TABLE 43. String and Character Escape Sequences

Escape	Description
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed
\r	Carriage return
\”	Double quote
\’	Single quote
\\	Backslash
\xxx	Character of octal value xxx
\uxxxx	Character of hexadecimal value xxxx

Operator Precedence

TABLE 44. Operator Precedence Table

Operator	Precedence Level	Fix	Associativity
[ ]	10	Right bracket	
.	10	Infix	Left
( )	10	Right bracket	
+	20	Prefix	Right
-	20	Prefix	Right
!	20	Prefix	Right
*	30	Infix	Left
/	30	Infix	Left
div	30	Infix	Left
mod	30	Infix	Left
+	40	Infix	Left
-	40	Infix	Left
member	45	Infix	Left
inside	45	Infix	Left
!inside	45	Infix	Left
directlyinside	45	Infix	Left
!directlyinside	45	Infix	Left
contain	45	Infix	Left
!contain	45	Infix	Left
directlycontain	45	Infix	Left
!directlycon-tain	45	Infix	Left
after	45	Infix	Left
!after	45	Infix	Left
directlyafter	45	Infix	Left
!directlyafter	45	Infix	Left
before	45	Infix	Left
!before	45	Infix	Left

TABLE 44. Operator Precedence Table

Operator	Precedence Level	Fix	Associativity
directlybefore	45	Infix	Left
!directlybefore	45	Infix	Left
overlap	45	Infix	Left
!overlap	45	Infix	Left
intersect	45	Infix	Left
without	45	Infix	Left
<	60	Infix	Left
<=	60	Infix	Left
>	60	Infix	Left
>=	60	Infix	Left
==	70	Infix	Left
!=	70	Infix	Left
and	80	Infix	Right
or	90	Infix	Right
=	100	Infix	Right
:=	100	Infix	Right
|	110	Infix	Right
?	110	Infix	Right

Note: Operators with a higher precedence level (smaller numeric values) bind tighter than those of a lower precedence level.

Operators

TABLE 45. WebL Operators

Operator	Description
!(x: bool): bool	Logical negation.
!=(x, y): bool	Value in-equality test. See "Value Equality" on page 31.
!after(p: piece, q: piece): pieceset **!after**(p: pieceset, q: piece): pieceset **!after**(p: piece, q: pieceset): pieceset **!after**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not after any element of *q*.
!before(p: piece, q: piece): pieceset **!before**(p: pieceset, q: piece): pieceset **!before**(p: piece, q: pieceset): pieceset **!before**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not precede any element of *q*.
!contain(p: piece, q: piece): pieceset **!contain**(p: pieceset, q: piece): pieceset **!contain**(p: piece, q: pieceset): pieceset **!contain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not contain any element of *q*.
!directlyafter(p: piece, q: piece): pieceset **!directlyafter**(p: pieceset, q: piece): pieceset **!directlyafter**(p: piece, q: pieceset): pieceset **!directlyafter**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not follow directly after any element of *q*.
!directlybefore(p: piece, q: piece): pieceset **!directlybefore**(p: pieceset, q: piece): pieceset **!directlybefore**(p: piece, q: pieceset): pieceset **!directlybefore**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not directly before any element of *q*.
!directlycontain(p: piece, q: piece): pieceset **!directlycontain**(p: pieceset, q: piece): pieceset **!directlycontain**(p: piece, q: pieceset): pieceset **!directlycontain**(p: pieceset, q: pieceset): pieceset	All the elements of p that do not directly contain any element of q.
!directlyinside(p: piece, q: piece): pieceset **!directlyinside**(p: pieceset, q: piece): pieceset **!directlyinside**(p: piece, q: pieceset): pieceset **!directlyinside**(p: pieceset, p: pieceset): pieceset	All the elements of *p* that are not directly inside any element of *q*.

TABLE 45. WebL Operators

Operator	Description
!**inside**(p: piece, q: piece): pieceset !**inside**(p: pieceset, q: piece): pieceset !**inside**(p: piece, q: pieceset): pieceset !**inside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are not located inside any element of *q*.
!overlap(p: piece, q: piece): pieceset **!overlap**(p: pieceset, q: piece): pieceset **!overlap**(p: piece, q: pieceset): pieceset **!overlap**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that do not overlap any element in *q*.
*(q1: piece, q2: piece): pieceset *(q: piece, s: pieceset): pieceset *(s: pieceset, q: piece): pieceset *(s1: pieceset, s2: pieceset): pieceset	Piece set intersection.
*(x: int, y: int): int *(x: int, y: real): real *(x: real, y: int): real *(x: real, y: real): real	Numeric multiplication.
*(x: set, y: set): set	Set intersection.
+(q1: piece, q2: piece): pieceset +(q: piece, s: pieceset): pieceset +(s: pieceset, q: piece): pieceset +(s1: pieceset, s2: pieceset): pieceset	Piece set union.
+(x: char, y: string): string +(x: char, y: char): string +(x: string, y: string): string +(x: string, y: char): string	String and character concatenation.
+(x: int, y: int): int +(x: int, y: real): real +(x: real, y: int): real +(x: real, y: real): real	Numeric addition *x + y*.
+(x: list, y: list): list	List concatenation.
+(x: set, y: set): set	Set union.
-(q1: piece, q2: piece): pieceset -(q: piece, s: pieceset): pieceset -(s: pieceset, q: piece): pieceset -(s1: pieceset, s2: pieceset): pieceset	Piece set difference.
-(x: int): int -(x: real): real	Numeric negation.

TABLE 45. WebL Operators

Operator	Description
-(x: int, y: int): int -(x: int, y: real): real -(x: real, y: int): real -(x: real, y: real): real	Numeric substraction.
-(x: set, y: set): set	Set exclusion.
.(x: object, y): any	Object field access.
/(x: int, y: int): int /(x: int, y: real): real /(x: real, y: int): real /(x: real, y: real): real	Numeric division.
== (x, y): bool	Value equality test. See "Value Equality" on page 31.
[](s: pieceset, i: int): piece	Indexing into a piece set. Pieces are numbered *0* to *Size - 1*.
[](x: list, i: int): any [](x: object, i): any [](x: string, i: int): char	List, object, and string indexing[a]. Elements in a list and string are numbered from *0* to *Size-1*.
after(p: piece, q: piece): pieceset **after**(p: pieceset, q: piece): pieceset **after**(p: piece, q: pieceset): pieceset **after**(p: pieceset, q: pieceset): pieceset	All the elements of p that are after any element of q.
before(p: piece, q: piece): pieceset **before**(p: pieceset, q: piece): pieceset **before**(p: piece, q: pieceset): pieceset **before**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that precede any element of *q*.
C(x: int, y: int): bool C(x: int, y: real): bool C(x: real, y: int): bool C(x: real, y: real): bool	Numerical comparison, where *C* is one of <, <=, >, or >=.
C(x: string, y: string): bool C(x: char, y: char): bool	Lexical comparison, where *C* is one of <, <=, >, or >=.
contain(p: piece, q: piece): pieceset **contain**(p: pieceset, q: piece): pieceset **contain**(p: piece, q: pieceset): pieceset **contain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that contain any element of *q*.

TABLE 45. WebL Operators

Operator	Description
directlyafter(p: piece, q: piece): pieceset **directlyafter**(p: pieceset, q: piece): pieceset **directlyafter**(p: piece, q: pieceset): pieceset **directlyafter**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that follow directly after any element of *q*.
directlybefore(p: piece, q: piece): pieceset **directlybefore**(p: pieceset, q: piece): pieceset **directlybefore**(p: piece, q: pieceset): pieceset **directlybefore**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are directly before any element of *q*.
directlycontain(p: piece, q: piece): pieceset **directlycontain**(p: pieceset, q: piece): pieceset **directlycontain**(p: piece, p: pieceset): pieceset **directlycontain**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that directly contain any element of *q*.
directlyinside(p: piece, q: piece): pieceset **directlyinside**(p: pieceset, q: piece): pieceset **directlyinside**(p: piece, q: pieceset): pieceset **directlyinside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are directly inside any element of *q*.
div(x: int, y: int): int	Whole division.
inside(p: piece, q: piece): pieceset **inside**(p: pieceset, q: piece): pieceset **inside**(p: piece, q: pieceset): pieceset **inside**(p: pieceset, q: pieceset): pieceset	All the elements of *p* that are located inside any element of q.
intersect(p: piece, q: piece): pieceset **intersect**(p: pieceset, q: piece): pieceset **intersect**(p: piece, q: pieceset): pieceset **intersect**(q: pieceset, p: pieceset): pieceset	All the elements of *p* that overlap an element in *q*, each of them repeatedly intersected with all overlapping elements in *q*.
member(x, s: set): bool **member**(x, l: list): bool **member**(x, o: object): bool	Set, list and object[b] membership test.
mod(x: int, y: int): int	*x* mod *y*.
or(x: bool, y: bool): bool **and**(x: bool, y: bool): bool	Logical operators (short-circuit evaluation).

TABLE 45. WebL Operators

Operator	Description
overlap(p: piece, q: piece): pieceset **overlap**(p: pieceset, q: piece): pieceset **overlap**(p: piece, q: pieceset): pieceset **overlap**(p: pieceset, p: pieceset): pieceset	All the elements of *p* that overlap any element in *q*.
without(p: piece, q: piece): pieceset **without**(p: pieceset, q: piece): pieceset **without**(p: piece, q: pieceset): pieceset **without**(p: pieceset, q: pieceset): pieceset	All the elements of *p* where overlaps with any element of *q* have been removed.

a. Right bracket fix operator of the form x[i].

b. Object membership test is based on object field names.

Functions

TABLE 46. Built-in Functions

Function	Description
Assert(x: bool)	Throws an assertion-failed exception if *x* is false.
BeginTag(q: piece): tag	Returns the begin tag of a piece.
Boolp(x): bool Charp(x): bool Funp(x): bool Intp(x): bool Listp(x): bool Methp(x): bool Objectp(x): bool Realp(x): bool Setp(x): bool Stringp(x): bool Pagep(x): bool Piecep(x): bool Tagp(x): bool Piecesetp(x): bool	Predicates that check if a value is of a specific type.
Call(cmd: string): string	Executes a shell command and returns the output written to standard out while the command is running. The command string may contain references to variables in lexical scope by writing *$var* or *${var}*. The value of these referenced variables are expanded before the command is executed.
Children(q: piece): pieceset	Returns a piece set consisting of all the direct children elements of *q* in the markup parse tree, unioned with pieces representing all the text segments in *q* (excluding all the nested text segments).
Clone(o: object, p: object, ...): object	Makes a new object by copying all the fields of the objects passed as arguments. Fields of *p* have precedence over fields of *o* (and so on).

TABLE 46. Built-in Functions

Function	Description
Content(p: page): piece	Returns a piece that encompasses the whole page *p*.
Content(q: piece): piece	Returns a piece inside *q*, representing everything that is inside *q* excluding the begin and end tag of *q*.
Delete(s: pieceset): nil Delete(q: piece): nil	Deletes *s* or *q* from the page by removing all the pieces from the page data structure.
Elem(p: page): pieceset	Returns all the elements in a page.
Elem(p: page, name: string): pieceset	Returns all the elements in page p with a specific name.
Elem(q: piece): pieceset	Returns all the elements contained (nested) in piece q.
Elem(q: piece, name: string): pieceset	Returns all the elements with a specific name contained in piece q.
EndTag(q: piece): tag	Returns the end tag of a piece.
Error(x, y, z, ...): nil	Prints arguments to standard error output.
ErrorLn(x, y, z, ...): nil	Prints arguments to standard error output followed by end-of-line.
Eval(s: string): any	Evaluates the WebL program coded in string *s*.
Exec(cmd: string): int	Executes a shell command and returns the exit code returned by the command. The command string may contain references to variables in lexical scope by writing *$var* or *${var}*. The value of these referenced variables are expanded before the command is executed.
Exit(errorcode: int)	Terminates the program with an errorcode.

TABLE 46. Built-in Functions

Function	Description
ExpandCharEntities(p: page, s: string): string	Expands the character entities (eg. "<", "&") in *s* to their Unicode character equivalents. The DTD of page *p* is used for the lookups.
ExpandCharEntities(s: string): string	Expands the character entities (eg. "<", "&") in *s* to their Unicode character equivalents. The HTML 4.0 DTD is used for the lookups.
DeleteField(o: object, fld): nil	Removes the field *fld* from the object *o*. Nothing happens if the field *fld* does not exist,
First(l: list): any	Returns the first element in a list.
Flatten(s: pieceset): pieceset	Returns a “flattened” piece set (without any overlapping) of all the parts of the page covered by *s*.
GC(): nil	Explicitly invokes the Java garbage collector.
GetURL(url: string): page	Uses the HTTP GET protocol to fetch the resource identified by the URL.
GetURL(url: string, params: {object,string}): page	The *params* object/string contains the parameters of a GET that includes a query.
GetURL(url: string, params: {object,string}, headers: object): page	The *headers* object specifies the additional headers to include in the GET request.
GetURL(url: string, params: {object,string}, headers: object. options: object): page	The *options* object allows, amongst other functions, the overridng of the MIME type and DTD to be used for parsing the page.
HeadURL(url: string): page	Uses the HTTP HEAD protocol to fetch the resource headers identified by the URL.
HeadURL(url: string, params: {object,string}): page	The *params* object contains the parameters of the HEAD request.

TABLE 46. Built-in Functions

Function	Description
HeadURL(url: string, params: {object,string}, headers: object): page	The headers object specifies the additional headers to include in the HEAD request.
InsertAfter(t: tag, q: piece): nil InsertAfter(t: tag, s: pieceset): nil	Inserts a copy of *q* after the tag *t*. Inserts copies of the elements of *s* after the tag *t*.
InsertBefore(t: tag, q: piece): nil InsertBefore(t: tag, s: pieceset): nil	Inserts a copy of *q* before the tag *t*. Inserts copies of the elements of *s* before the tag *t*.
Markup(p: page): string	Turns a page object back into a string.
Markup(q: piece): string	Turns a piece object back into a string.
Name(q: piece): string	Returns the name of a piece.
Native(classname: string): fun	Loads a WebL function[a] implemented in Java.
NewNamedPiece(name: string, q: piece): piece	Equivalent to *NewNamedPiece(name, BeginTag(q), EndTag(q))*.
NewNamedPiece(name: string, t1: tag, t2: tag): piece	Returns a new named piece starting before *t1* and ending after *t2*.
NewPage(s: string, mimetype: string): page	Parses the string *s* with the mimetype indicated markup parser and returns a page object.
NewPiece(q: piece): piece	Equivalent to *NewPiece(BeginTag(q), EndTag(q))*.
NewPiece(s: string, mimetype: string): piece	Equivalent to *Content(NewPage(s, mimetype))*.
NewPiece(t1: tag, t2: tag): piece	Returns a new unnamed piece starting before *t1* and ending after *t2*.
NewPieceSet(s: set): pieceset	Converts a set of pieces into a piece set. Throws an *EmptySet* exception should *s* be empty.
NewPieceSet(p: page): pieceset	Returns an empty pieceset associated with with page *p*.

TABLE 46. Built-in Functions

Function	Description
Page(q: piece): page	Returns the page a piece belongs to.
Page(t: tag): page	Returns the page a tag belongs to.
Para(p: page, paraspec: string): pieceset	Extracts the paragraphs in *p* according to the paragraph terminator specification *paraspec*. See "Paragraph search" on page 75.
Para(p: piece, paraspec: string): pieceset	Extracts the paragraphs in *p* according to the paragraph terminator specification *paraspec*. See "Paragraph search" on page 75.
Parent(q: piece): piece	Returns the element in which *q* is nested (direct parent in the parse tree).
Pat(p: page, regexp: string): pieceset	Returns all the occurrences of a regular expression pattern in page p.
Pat(q: piece, regexp: string): pieceset	Returns all the occurrences of a regular expression pattern located inside the piece *q*.
PostURL(url: string): page	Uses the HTTP POST protocol to fetch the resource identified by the URL.
PCData(p: page): pieceset	Returns the "parsed character data" of the page. This corresponds to the individual seqences of text on the page, as delimited by markup tags.
PCData(p: piece): pieceset	Returns the "parsed character data" of the piece. This corresponds to the individual seqences of text inside the piece, as delimited by markup tags.
PostURL(url: string, params: {object,string}): page	The *params* object/string contains the parameters of a POST to fill in a web form.
PostURL(url: string, params: {object,string}, headers: object): page	The *headers* object specifies the additional headers to include in the POST request.

TABLE 46. Built-in Functions

Function	Description
PostURL(url: string, params: {object,string}, headers: object. options: object): page	The *options* object allows, amongst other functions, the overridng of the MIME type and DTD to be used for parsing the page.
Pretty(p: page): string	Returns a pretty-printed version of the page.
Pretty(q: piece): string	Returns a pretty-printed version of a piece.
Print(x, y, z, ...): nil	Prints arguments to standard output.
PrintLn(x, y, z, ...): nil	Prints arguments to standard output followed by end-of-line.
ReadLn(): string	Reads a line from standard input (throws away the end-of-line character).
Replace(a: pieceset, b: pieceset): nil	Replaces each piece set of *a* with copies of all the elements of *b*.
Rest(l: list): list	Returns a list of all list elements except the first element.
Retry(x): any	Executes expression *x* and returns its value. In case *x* throws an exception, *x* is re-executed as many times as needed until it is successful.
Select(l: list, from: int, to: int): list	Extracts a sublist of *l* starting at element number *from* and ending at element number *to* (exclusive).
Select(s: set, f: fun): set Select(l: list, f: fun): list Select(p: pieceset: f: fun): pieceset	Maps sets, lists, and piecesets to sets, lists, and piecesets respectively according to a membership function *f*. Function *f* must have a single argument and must return a boolean value indicating whether the actual argument is to be included in the set, list or pieceset.
Select(s: string, from: int, to: int): string	Extracts a substring of *s* starting at character number *from* and ending at character number *to* (exclusive).

TABLE 46. Built-in Functions

Function	Description
Seq(p: page, pattern: string): pieceset	Matches all the occurrences of a sequence of elements identified by *pattern*. See "PCData search" on page 73.
Seq(p: piece, pattern: string): pieceset	Matches all the occurrences of a sequence of elements identified by *pattern* inside the piece *p*. See "PCData search" on page 73.
Sign(x: int): int Sign(x: real): int	Returns -1, 0, +1 if x < 0, x = 0, and x > 0 respectively.
Size(l: list): int	Returns the number of elements in a list.
Size(s: set): int	Returns the number of elements in a set.
Size(s: string): int	Returns the number of characters in a string.
Size(p: pieceset): int	Returns the number of pieces belonging to *p*.
Sleep(ms: int): nil	Suspends thread execution for the specified number of milliseconds.
Sort(l: list, f: fun): list	Sorts the elements of *l* according to the comparison function *f*. The function *f* needs to take two formal arguments and return -1, 0, or +1 if the actual arguments are less, equal, or more than each other.
Stall()	Program goes to sleep forever.
Text(p: page): string	Returns the text (sans tags) of a page.
Text(q: piece): string	Returns the text (sans tags) of a piece.

TABLE 46. Built-in Functions

Function	Description
Text(q: piece, insertspaces: boolean): string	Returns the text (sans tags) of a piece. When *insertspaces* is true, each HTML tag is mapped into a space and inserted into the result string (inline tags like “b”, “i”, “em”, etc. are ignored and not mapped into spaces). This option is useful to correctly identify word boundaries, for example to prevent words flowing together in a case like “ - wordA - wordB ”.
Throw(o: object)	Generates an exception.
Time(x): int	Returns the time (in milliseconds) it takes to evaluate the expression *x*.
Timeout(ms: int, x): any	Performs the expression *x* and returns its value. If the evaluation takes more than the specified amount of time (in milliseconds), an exception is thrown instead.
ToChar(c: char): char	No operation.
ToChar(i: int): char	Converts an integer to the equivalent Unicode character.
ToInt(c: char): int	Returns the Unicode character number of a char.
ToInt(i: int): int	No operation.
ToInt(r: real): int	Rounds a real value down to an integer.
ToInt(s: string): int	Converts a string to the numeric equivalent.
ToList(s: set): list ToList(l: list): list ToList(s: string): list ToList(o: object): list ToList(p: pieceset): list	Enumerates all the elements of the argument and returns a list. (See “Every Statement” on page 38.)
ToReal(c: char): real	Same as *ToReal(ToInt(c))*.
ToReal(i: int): real	Converts an integer to a real.

TABLE 46. Built-in Functions

Function	Description
ToReal(r: real): real	No operation.
ToReal(s: string): real	Converts a string to a real value.
ToSet(s: set): set ToSet(l: list): set ToSet(s: string): set ToSet(o: object): set ToSet(p: pieceset): set	Enumerates all the elements of the argument and returns a set. (See "Every Statement" on page 38.)
ToString(x): string	Converts a value to its string representation.
Trap(x):object	Executes *x* and returns the exception object that was caught. In case no exception is thrown in *x*, *nil* is returned. In addition, the exception object contains a field *trace* that has extra information why the exception occurred. This information is useful for logging unexpected exception events in your WebL programs.
Type(x): string	Returns the type of *x* (nil, int, real, bool, char, string, meth, fun, set, list, object, page, piece, pieceset, tag).

a. The class indicated must be a subclass of *webl.lang.expr.AbstractFunExpr*

Exceptions

Exceptions typically indicate unexpected situations occuring during program execution. Exceptions are caught with the *try* statement (See “Try Statement” on page 36) and generated with the *Throw* built-in function. Processing exceptions require knowledged about the format of exception objects, in particularly the *type* of the exception, which allows you to distinguish between the possible situations that occurred.

Table 47 lists the exceptions thrown by the built-in WebL functions. By convention, the exception type (eg *ArgumentError* etc.) is indicated by the *type* field of the exception object. Also by convention, the *msg* field of the exception object gives information on why the exception occured.

Operators and statements can also generate exceptions, as explained in the following paragraphs.

All operators will throw an *OperandMismatch* exception in case the operands to the operator are not of the expected value type.

Function or method application (eg calling a function or method) can throw the following exceptions:

- *NoSuchField* - Object does not have such field.
- *NotAFunctionOrMethod* - Left-hand side is not callable.
- *NotAnObject* - Left hand side is not an object.
- *ArgumentError* - Number of actual and formal arguments do not match.

Variable assigment with "=" can throw the following exceptions:

- *FieldError* - Unknown field or illegal field assignment.
- *NotAnObject* - Left hand side is not an object field.
- *NotAVariable* - Left hand side is not a variable.

Field definition with ":=" can throw the following exceptions:

- *FieldDefinitionError* - Could not define field.
- *NotAnObject* - Left hand side is not an object.

Indexing into a type with "[ ]" or "." can throw the following exceptions:

- *IndexRangeError* - Index is out of range.
- *ArgumentError* - Index is not of the expected type.
- *NoSuchField* - Object does not have such field.
- *NotAnObject* - Left hand side is not an object.

The *if*, *repeat*, *while* and *catch* statements will throw a *GuardError* exception if the guard expression does not return a boolean value type.

The *every* statement will throw a *NotEnumerable* exception if the object does not have enumerable contents.

The *lock* statement will throw a *NotAnObject* exception if an attempt is made to lock on a non-object value type.

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Assert(x: bool): nil	ArgumentError - Incorrect or wrong number of arguments AssertFailed - Assertion failed
BeginTag(q: piece): tag	ArgumentError - Incorrect or wrong number of arguments
Boolp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Exec(cmd: string): int	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Charp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Children(q: piece): pieceset	ArgumentError - Incorrect or wrong number of arguments
Clone(o: object, p: object, ...): object	ArgumentError - Incorrect or wrong number of arguments
Content(p: page): piece Content(q: piece): piece	NoContent - Page or piece has no content ArgumentError - Incorrect or wrong number of arguments
Delete(s: pieceset): nil Delete(q: piece): nil	ArgumentError - Incorrect or wrong number of arguments
Elem(p: page): pieceset Elem(p: page, name: string): pieceset Elem(q: piece): pieceset Elem(q: piece, name: string): pieceset	ArgumentError - Incorrect or wrong number of arguments
EndTag(q: piece): tag	ArgumentError - Incorrect or wrong number of arguments
Error(x, y, z, ...): nil	No exceptions are thrown
ErrorLn(x, y, z, ...): nil	No exceptions are thrown

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Eval(s: string): any	ArgumentError - Incorrect or wrong number of arguments SyntaxError - Cannot evaluate due to syntax error in argument IOException - An IO exception occurred during function execution ReturnException - A return statement was executed outside of a function or method while executing the argument
Exec(cmd: string): int	ArgumentError - Incorrect or wrong number of arguments
Exit(errorcode: int): nil	ArgumentError - Incorrect or wrong number of arguments
ExpandCharEntities(p: page, s: string): string ExpandCharEntities(s: string): string	ArgumentError - Incorrect or wrong number of arguments IOException - An IO exception occurred during function execution
First(l: list): any	ArgumentError - Incorrect or wrong number of arguments EmptyList - Cannot apply first to an empty list

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Flatten(s: pieceset): pieceset	ArgumentError - Incorrect or wrong number of arguments
Funp(x): bool	ArgumentError - Incorrect or wrong number of arguments
GC(): nil	ArgumentError - Incorrect or wrong number of arguments
GetURL(url: string): page) GetURL(url: string, params: {object, string}): page) GetURL(url: string, params: {object, string}, headers: object): page) GetURL(url: string, params: {object, string}, headers: object, options: object): page)	ArgumentError - Incorrect or wrong number of arguments NetException - Fetch failed, "statuscode" field of the exception object indicates the reason
GetURL(url: string): page) GetURL(url: string, params: {object, string}): page) GetURL(url: string, params: {object, string}, headers: object): page)	ArgumentError - Incorrect or wrong number of arguments NetException - Fetch failed, "statuscode" field of the exception object indicates the reason
InsertAfter(t: tag; q: piece): nil InsertAfter(t: tag; s: pieceset): nil	ArgumentError - Incorrect or wrong number of arguments
InsertBefore(t: tag; q: piece): nil InsertBefore(t: tag; s: pieceset): nil	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Intp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Listp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Markup(P: page): string Markup(q: piece): string	ArgumentError - Incorrect or wrong number of arguments
Methp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Name(q: piece): string	ArgumentError - Incorrect or wrong number of arguments
Native(classname: string): fun	ArgumentError - Incorrect or wrong number of arguments NativeCodeImportError - Class instantiation failed, access denied, no such method, or not a sub-class of built-in
NewNamedPiece(name: string, t1: tag, t2: tag): piece NewNamedPiece(name: string, q: piece): piece	ArgumentError - Incorrect or wrong number of arguments NotSamePage - The tag arguments to the function do not belong to the same page

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
NewPage(s: string, mimetype: string): page	ArgumentError - Incorrect or wrong number of arguments NetException - Fetch failed, "statuscode" field of the exception object indicates the reason
NewPiece(q: piece): piece NewPiece(s: string, mimetype: string): piece NewPiece(t1: tag, t2: tag): piece	ArgumentError - Incorrect or wrong number of arguments NotSamePage - The tag arguments to the function do not belong to the same page
NewPieceSet(p: page): pieceset	ArgumentError - Incorrect or wrong number of arguments NotAPiece - The set argument to NewPieceSet must only contain pieces EmptySet - The set argument to NewPieceSet must only contain pieces belonging to the same page NotSamePage - The set argument to NewPieceSet must only contain pieces belonging to the same page
Objectp(x): bool	ArgumentError - Incorrect or wrong number of arguments
PCData(p: page): pieceset PCData(p: piece): pieceset	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Page(q: piece): page Page(t: tag): page	ArgumentError - Incorrect or wrong number of arguments
Pagep(x): bool	ArgumentError - Incorrect or wrong number of arguments
Para(p: page, paraspec: string): pieceset Para(q: piece, paraspec: string): pieceset	ArgumentError - Incorrect or wrong number of arguments
Parent(q: piece): piece	ArgumentError - Incorrect or wrong number of arguments
Pat(p: page, regexp: string): pieceset Pat(q: piece, regexp: string): pieceset	ArgumentError - Incorrect or wrong number of arguments MalformedPattern - Illegal regular expression passed to Pat function
PieceSetp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Piecep(x): bool	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
PostURL(url: string): page) PostURL(url: string, params: {object, string}): page)	ArgumentError - Incorrect or wrong number of arguments
PostURL(url: string, params: {object, string}, headers: object): page) PostURL(url: string, params: {object, string}, headers: object, options: object): page)	NetException - Fetch failed, "statuscode" field of the exception object indicates the reason IOException - An IO exception occurred during function execution
Pretty(p: page): string Pretty(q: piece): string	ArgumentError - Incorrect or wrong number of arguments
Print(x, y, z, ...): nil	No exceptions are thrown
PrintLn(x, y, z, ...): nil	No exceptions are thrown
ReadLn(): string	ArgumentError - Incorrect or wrong number of arguments IOException - An IO exception occurred during function execution
Realp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Replace(a: pieceset, b: pieceset): nil	ArgumentError - Incorrect or wrong number of arguments
Rest(l: list): list	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Retry(x): any	ArgumentError - Incorrect or wrong number of arguments
Select(s: set: f: fun): set Select(l: list, f: fun): list Select(p: pieceset, f: fun): pieceset Select(s: string, from: int, to: int): string	ArgumentError - Incorrect or wrong number of arguments
Seq(p: page, pattern: string): pieceset Seq(q: piece, pattern: string): pieceset	ArgumentError - Incorrect or wrong number of arguments IndexRangeError - Index into list or string out of bounds FunctionReturnTypeNotBoolean - Function argument to Select did not return a boolean value
Setp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Sign(x: int): int Sign(x: real): int	ArgumentError - Incorrect or wrong number of arguments
Size(s: set):int Size(s: string): int Size(l: list): int	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Sleep(ms: int): nil	ArgumentError - Incorrect or wrong number of arguments Interrupted - Sleep function interrupted
Sort(l: list, f: fun): list	ArgumentError - Incorrect or wrong number of arguments FunctionReturnTypeNotInteger - Function argument to Sort did not return an integer value
Stall()	ArgumentError - Incorrect or wrong number of arguments
Stringp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Tagp(x): bool	ArgumentError - Incorrect or wrong number of arguments
Text(p: page): string Text(q: piece): string	ArgumentError - Incorrect or wrong number of arguments
Throw(o: object)	ArgumentError - Incorrect or wrong number of arguments
Time(x): int	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Timeout(ms: int, x): any	ArgumentError - Incorrect or wrong number of arguments Timeout - A time out occurred
ToChar(c: char): char ToChar(i: int): char	ArgumentError - Incorrect or wrong number of arguments
ToInt(c: char): int ToInt(i: int): int ToInt(r: real): int ToInt(s: string): int	ArgumentError - Incorrect or wrong number of arguments
ToList(s: set): list ToList(l: list): list ToList(s: string): list ToList(o: object): list ToList(p: pieceset): list	ArgumentError - Incorrect or wrong number of arguments
ToReal(c: char): real ToReal(i: int): real ToReal(r: real): real ToReal(s: string): real	ArgumentError - Incorrect or wrong number of arguments
ToSet(s: set): set ToSet(l: list): set ToSet(s: string): set ToSet(o: object): set ToSet(p: pieceset): set	ArgumentError - Incorrect or wrong number of arguments
ToString(x): string	ArgumentError - Incorrect or wrong number of arguments

TABLE 47. Exceptions thrown by the built-in functions

Function	Exceptions
Trap(x): object	ArgumentError - Incorrect or wrong number of arguments
ToString(x): string	ArgumentError - Incorrect or wrong number of arguments

Regular Expressions

Here we summarize the syntax of Perl5 regular expressions, all of which are supported by the WebL. However, for a definitive reference, you should consult the *perlre* man page that accompanies the Perl5 distribution and also the book Programming Perl, 2nd Edition from O'Reilly & Associates. We need to point out here that for efficiency reasons the character set operator [...] is limited to work on only ASCII characters (Unicode characters 0 through 255). Other than this restriction, all Unicode characters should be useable in the package's regular expressions.

Perl5 regular expressions consist of:

- Alternatives separated by |
- Quantified atoms (Table 48)
- Atoms

  Regular expression within parentheses, character classes (e.g., [abcd]), ranges (e.g. [a-z]), and the patterns in Table 50. Special backslashed characters work within a character class (except for backreferences and boundaries). \b is backspace inside a character class. Any other backslashed character matches itself. Expressions within parentheses are matched as subpattern groups and saved for use by certain methods.

TABLE 48. Quantified Atoms

Pattern	Description
{n,m}	Match at least n but not more than m times.
{n,}	Match at least n times.
{n}	Match exactly n times.
*	Match 0 or more times.
+	Match 1 or more times.
?	Match 0 or 1 times.

By default, a quantified subpattern is greedy. In other words, it matches as many times as possible without causing the rest of the pattern not to match. To change the quantifiers to match the minimum number of times possible, without causing the rest of the pattern not to match, you may use a "?" right after the quantifier (Table 49). Perl5 extended regular expressions are fully supported (See Table 51).

Regular Expression Tips. Combining regular expresions and WebL code might sometimes be a little confusing. The following tips might help:

- When possible, write WebL regular expressions in single-back quotes e.g. `'ab\nc'`. This will switch off escape character expansion, and prevent WebL from complaining about illegal escape sequences like "\d".
- When matching URLs, keep in mind that "." and "?" do not have a literal meaning in regular expressions. Use the "[]" character classes to match these symbols, e.g. write "www[.]xyz[.]com" instead of "www.xyz.com".

TABLE 49. Quantified Atoms with Minimal Matching

Pattern	Description
{n,m}?	Matches at least n but not more than m times.
{n,}?	Matches at least n times.
{n}?	Matches exactly n times.
*?	Matches 0 or more times.
+?	Matches 1 or more times.
??	Matches 0 or 1 times.

TABLE 50. Atoms

Pattern	Description
.	Matches everything except \n.
^	Null token matching the beginning of a string or line (i.e. the position right after a newline or right before the beginning of a string).
$	Null token matching the end of a string or line (i.e. the position right before a newline or right after the end of a string).
\b	Null token matching a word boundary (\w on one side and \W on the other).
\B	Null token matching a boundary that is not a word boundary.
\A	Matches only at beginning of string.
\Z	Matches only at end of string (or before newline at the end).
\n	Newline.
\r	Carriage return.
\t	Tab.
\f	Formfeed.
\d	Digit [0-9].
\D	Non-digit [^0-9].
\w	Word character [0-9a-zA-Z].
\W	Non-word character [^0-9a-zA-Z].
\s	A whitespace character [\t\n\r\f].
\S	A non-whitespace character [^ \t\n\r\f].
\xnn	Hexadecimal representation of character.
\cD	Matches the corresponding control character.
\nn or \nnn	Octal representation of character unless a backreference.
\1, \2, \3, etc.	Matches whatever the first, second, third, etc. parenthesized group matched. This is called a *backreference*. If there is no corresponding group, the number is interpreted as an octal representation of a character.
\0	Matches null character.

TABLE 51. Perl5 Extended Regular Expressions

Extended Pattern	Description
(?#text)	An embedded comment causing text to be ignored.
(?:regexp)	Groups things like "()" but does not cause the group match to be saved.
(?=regexp)	A zero-width positive lookahead assertion. For example, \w+(?=\s) matches a word followed by whitespace, without including whitespace in the match result.
(?!regexp)	A zero-width negative lookahead assertion. For example foo(?!bar) matches any occurrence of "foo" that is not followed by "bar". Remember that this is a zero-width assertion, which means that a(?!b)d will match ad because a is followed by a character that is not b (the d) and a d follows the zero-width assertion.
(?imsx)	One or more embedded pattern-match modifiers. *i* enables case insensitivity, *m* enables multiline treatment of the input, *s* enables single line treatment of the input, and *x* enables extended whitespace comments.