Volume 2 / Modules and Services

AFNIX Writing System

Revision 3.8

Amaury C. Darsch

Visit http://www.afnix.org

Contents

Part 1

Modules

CHAPTER 1

Standard Graph Module

The *Standard Graph* module is an original implementation dedicated to the graph modeling and manipulation. At the heart of this module is the concept of edges and vertices. The module also provides support for automaton.

1. Graph concepts

The *afnix-gfx* module provides the support for manipulating graphs. Formally a graph is a collection of edges and vertices. In a normal graph, an edge connects two vertices. On the other hand, a vertex can have several edges. When an edge connects several vertices, it is called an hyperedge and the resulting structure is called an hypergraph.

1.1. Edge class. The *Edge* class is a class used for a graph construction in association with the *Vertex* class. An edge is used to connect vertices. Normally, an edge connects two vertices. The number of vertices attached to an edge is called the cardinality of that edge. When the edge cardinality is one, the edge is called a self-loop. This mean that the edge connects the vertex to itself. This last point is merely a definition but present the advantage of defining an hyperedge as a set of vertices.

1.2. Vertex class. The *Vertex* is the other class used for the graph construction. and operates with the *edge* class. A vertex is used to reference edges. the number of edges referenced by a vertex is called the degree of that vertex.

1.3. Graph. The *Graph* class is class that represent either a graph or a hypergraph. By definition, a graph is collection of edges and vertices. There are numerous property attached to graph. Formally, a graph consists of a set of edges, a set of vertices and the associated endpoints. However, the implementation is designed in a way so that each edge and vertex carry its associated objects. This method ensures that the graph is fully defined by only its two sets.

2. Graph construction

The graph construction is quite simple and proceed by adding edges and vertices. The base system does not enforce rules on the graph structure. it is possible to add con connected vertices as well as unreferenced edges.

2.1. Edge construction. An edge is constructed by simply invoking the default constructor. Optionally, a client object can be attached to the edge.

```
1 # create a default edge
2 const edge (afnix:gfx:Edge)
3 # create an edge with a client object
4 const hello (afnix:gfx:Edge "hello")
```

The *edge-p* predicate can be used to check for the object type. When an edge is created with client object, the *get-client* method can be used to access that object.

2.2. Vertex construction. A vertex is constructed a way similar to the *Edge>* object. The vertex is constructed by simply invoking the default constructor. Optionally, a client object can be attached to the edge.

```
# create a default vertex
const vrtx (afnix:gfx:Vertex)
# create an vertex with a client object
const world (afnix:gfx:Vertex "world")
```

The *vertex-p* predicate can be used to check for the object type. When a vertex is created with a client object, the *get-client* method can be used to access that object.

2.3. Graph construction. A graph is constructed by simply adding edges and vertices to it. The *graph-p* predicate can be use to assert the graph type. the graph class also supports the concept of client object which can be attached at construction or with the *set-client* method.

```
const graph (afnix:gfx:Graph)
```

The *add* method can be used to add edges or vertices to the graph. The important point is that during the construction process, the graph structure is updated with the proper number of edge and vertices.

```
# create a graph
const g (afnix:gfx:Graph)
assert true (afnix:gfx:graph-p g)
# create an edge and add vertices
const edge (afnix:gfx:Edge)
edge:add (afnix:gfx:Vertex "hello")
edge:add (afnix:gfx:Vertex "world")
assert 2 (edge:degree)
# add the edge to the graph and check
g:add edge
assert 1 (g:number-of-edges)
assert 2 (g:number-of-vertices)
# check for nodes and edges
assert true (afnix:gfx:edge-p (g:get-edge 0))
assert true (afnix:gfx:vertex-p (g:get-vertex 0))
assert true (afnix:gfx:vertex-p (g:get-vertex 1))
```

CHAPTER 2

Standard Graph Reference

1. Object State

The *State* class is a class the graph object library. The state is used to model vertex and edge and can be used seldom in structure like finite automaton.

1.1. Predicate.

- state-p

1.2. Inheritance.

- Serial

1.3. Constructors.

- *State → (none)*
  The *State* constructor create an empty state.
- *State → (Object)*
  The *State* constructor create a state edge with a client object.

1.4. Methods.

- *clear → none (none)*
  The *clear* method clear the state marker.
- *get-index → Integer (none)*
  The *get-idex* method returns the state index.
- *set-idnex → none (Integer)*
  The *set-index* method sets the state index.
- *get-client → Object (none)*
  The *get-client* method returns the state client object. If the client object is not set, nil is returned.
- *set-client → Object (Object)*
  The *set-client* method sets the state client object. The object is returned by this method.

2. Object Edge

The *Edge* class is a class used for a graph construction in association with the *Vertex* class. An edge is used to connect vertices. Normally, an edge connects two vertices. The number of vertices attached to an edge is called the cardinality of that edge. A client object can also be attached to the class.

2.1. Predicate.

- edge-p

2.2. Inheritance.

- State, Collectable

2.3. Constructors.

- *Edge → (none)*
  The *Edge* constructor create an empty edge.
- *Edge → (Object)*
  The *Edge* constructor create an edge with a client object.

2.4. Methods.

- *reset → none (none)*
  The *reset* method reset all vertices attached to the edge.
- *cardinality → Integer (none)*
  The *cardinality* method returns the cardinality of the edge. The cardinality of an edge is the number of attached vertices.
- *add → Vertex (Vertex)*
  The *add* method attach a vertex to this edge. The method return the argument vertex.
- *get → Vertex (Integer)*
  The *get* method returns the attached vertex by index. If the index is out-of range, and exception is raised.

3. Object Vertex

The *Vertex* class is a class used for a graph construction in association with the *Edge* class. An vertex is an edge node. The number of edges referenced by a vertex is called the degree of that vertex. A client object can also be attached to the object.

3.1. Predicate.

- vertex-p

3.2. Inheritance.

- State, Collectable

3.3. Constructors.

- *Vertex → (none)*
  The *Vertex* constructor create an empty vertex.
- *Vertex → (Object)*
  The *Vertex* constructor create a vertex with a client object.

3.4. Methods.

- *reset → none (none)*
  The *reset* method reset all edges attached to the vertex.
- *degree → Integer (none)*
  The *degree* method returns the degree of the vertex. The degree of a vertex is the number of referenced edges.
- *add → Edge (Edge)*
  The *add* method references an edge with this vertex. The method return the argument edge.
- *get → Edge (Integer)*
  The *get* method returns the referenced edge by index. If the index is out-of range, and exception is raised.

4. Object Graph

The *Graph* object is a general graph class that manages a set of edges and vertices. The graph operates by adding edges and vertices to it.

4.1. Predicate.

- graph-p

4.2. Inheritance.

- Object

4.3. Constructors.

- *Graph → (none)*
  The *Graph* constructor create an empty graph.

4.4. Methods.

- *reset → none (none)*
  The *reset* method reset the graph
- *clear → none (none)*
  The *clear* method clear the graph
- *add → Object (Vertex—Edge)*
  The *add* method adds a vertex or an edge to the graph. When adding an edge, the methods check that the source and target vertices are also part of the graph.

CHAPTER 3

Standard Telecom Module

The *Standard Telecom* module is an original implementation of various standards managed by the International Telecommunictaion Union (ITU). At the heart of this module is the *Abstract Syntax Notation* (ASN.1) which is widely used to model data records and store certificates.

1. Abstract syntax notation

The abstract syntax notation (ASN.1) is standardized by the ITU to express a normal form of communication. The ASN.1 is in fact the de-facto standard for representing X509 certificate and is the only justification to provide the support for such complex representation.

1.1. Encoding rules. This implementation supports all encoding forms as defined by the ITU, namely the *Basic Encoding Rule* (BER), the *Canonical Encoding Rule* (CER) and the *Distinguished Encoding Rule* (DER). The DER form is by far the most widely used.

1.2. ASN objects. All objects as defined by the ITU are supported in this implementation, including the ability to create custom OID.

1.3. Using ASN.1 objects. Using ASN.1 object is particularly straightfoward. One can directly creates a particular object by invoking the appropriate constructor.

```
1 # create an asn boolean node
2 trans abn (afnix:itu:AsnBoolean true)
```

Object	Description
AsnBoolean	Boolean primitive
AsnInteger	Integer primitive
AsnBits	Bit string
AsnOctets	Octet string
AsnBmp	Bmp string
AsnIas	IA5 string
AsnNums	Numeric string
AsnPrts	Printable string
AsnUtfs	Unicode string
AsnUnvs	Universal string
AsneNull	Null primitive
AsneEoc	End-of-Content primitive
AsnGtm	Generalized time primitive
AsnUtc	Utc time primitive
AsnSequence	Asn node Sequence
AsnSet	Asn node Set
AsnOid	Asn object identifier Set
AsnRoid	Asn object relative identifier Set

```
# check the node type
assert true (afnix:itu:asn-node-p abn)
assert true (afnix:itu:asn-boolean-p abn)
```

Writing the object can be done into a buffer or an output stream. Note that the default encoding is the DER encoding.

```
# write into a buffer
trans buf (Buffer)
abn:write buf
# check the buffer content
assert "0101FF" (buf:format)
```

Building an ASN.1 representation can be achieved by parsing a buffer or an input stream. This is done by filling a buffer and requesting a buffer node mapping.

```
# parse the buffer and check
const anb (afnix:itu:AsnBuffer buf)
# map the node to a boolean
trans abn (anb:node-map)
# check the node
assert true (afnix:itu:asn-node-p abn)
assert true (afnix:itu:asn-boolean-p abn)
```

With more complex structure, it is likely that a sequence object will be returned by the buffer node mapper. Once the sequence object is created, each node can be accessed by index like any other container.

CHAPTER 4

Standard Telecom Reference

1. Object AsnNode

The *AsnNode* class is the base class used to represent the asn tree. The structure of the node is defined in ITU-T X.690 recommendation. This implementation supports 64 bits tag number with natural machine length encoding. The Canonical Encoding Rule (CER) and Distinguished Encoding Rule (DER) are defined by the class. Since ASN.1 provides several encoding schemes, the class is designed to be as generic as possible but does not provides the mechanism for changing from one representation to another although it is perfectly valid to read a DER representation and write it in the CER form.

1.1. Predicate.

- asn-node-p

1.2. Inheritance.

- Object

1.3. Constants.

- *BER → ()*
  The *BER* constant defines the *Basic Encoding Rule* node encoding.
- *CER → ()*
  The *CER* constant defines the *Canonical Encoding Rule* node encoding.
- *DER → ()*
  The *DER* constant defines the *Distinguished Encoding Rule* node encoding.
- *UNIVERSAL → ()*
  The *UNIVERSAL* constant defines the node universal class.
- *APPLICATION → ()*
  The *APPLICATION* constant defines the node application class.
- *CONTEXT-SPECIFIC → ()*
  The *CONTEXT-SPECIFIC* constant defines the node context specific class.
- *PRIVATE → ()*
  The *PRIVATE* constant defines the node private class.

1.4. Methods.

- *reset → none (none)*
  The *reset* method reset a node to its default value.
- *length → Integer (none)*
  The *length* method returns the total node length in bytes.
- *get-class → UNIVERSAL—APPLICATION—CONTEXT-SPECIFIC—PRIVATE (none)*
  The *get-class* method returns the node class.
- *primitive-p → Boolean (none)*
  The *primitive-p* returns true if the node is a primitive.
- *constructed-p → Boolean (none)*
  The *constructed-p* returns true if the node is a constructed node.
- *get-tag-number → Integer (none)*
  The *get-tag-number-p* returns node tag number.
- *get-content-length → Integer (none)*
  The *get-content-length-p* returns node content length.
- *write → none (none—OutputStream—Buffer)*
  The *write* method write the asn node contents as well as the child nodes to an output stream argument or a buffer. Without argument, the node is written to the interpreter output stream. With one argument, the node is written to the specified stream or buffer.

2. Object AsnOctets

The *AsnOctets* class is the asn object class that encodes the octet string type. This type can be encoded either as a primitive or as constructed at sender's option. In CER form, the primitive form is used when the content length is less than 1000 octets, and the constructed form is used otherwise. The DER form will always use the primitive form.

2.1. Predicate.

- asn-octets-p

2.2. Inheritance.

- AsnNode

2.3. Constructors.

- *AsnOctets → (none)*
  The *AsnOctets* constructor creates a default asn octets string node.
- *AsnOctets → (String—Buffer)*
  The *AsnOctets* constructor creates an asn octets string node by string of buffer object.

2.4. Methods.

- *to-buffer → Buffer (none)*
  The *to-buffer* method returns a *Buffer* object as an octet string representation.

3. Object AsnBuffer

The *AsnBuffer* class is the asn object class that provides a generic implementation of an asn structure. The class acts as a simple encoder and decoder with special facilities to retarget the buffer content.

3.1. Predicate.

- asn-buffer-p

3.2. Inheritance.

- AsnNode

3.3. Constructors.

- *AsnBuffer → (none)*
  The *AsnBuffer* constructor creates a default asn buffer node.
- *AsnBuffer → (InputStream—Buffer—Bitset)*
  The *AsnBuffer* constructor creates an asn buffer node from an input stream, a buffer or a bitset.

3.4. Methods.

- *reset → none (none)*
  The *reset* method reset the buffer.
- *parse → Boolean (InputStream—Buffer—Bitset)*
  The *parse* method parse a node represented by an input stream, a buffer or a bitset.
- *node-map → AsnNode (none)*
  The *node-map* method returns a node mapping of this buffer.
- *get-content-buffer → Buffer (none)*
  The *get-content-buffer* method returns the asn buffer content as a buffer object.

4. Object AsnNull

The *AsnNull* class is the asn object class that encodes the null primitive. This primitive has a unique encoding. The length is always 0 and there is no content octet.

4.1. Predicate.

- asn-null-p

4.2. Inheritance.

- AsnNode

4.3. Constructors.

- *AsnNull* → *(none)*
  The *AsnNull* constructor creates a default asn null node.

5. Object AsnEoc

The *AsnEoc* class is the asn object class that encodes the eoc or end-of-content primitive. This primitive is almost never used but its encoding is used with the indefinite length encoding.

5.1. Predicate.

- asn-eoc-p

5.2. Inheritance.

- AsnNode

5.3. Constructors.

- *AsnEoc* → *(none)*
  The *AsnEoc* constructor creates a default asn eoc node.

6. Object AsnBoolean

The *AsnBoolean* class is the asn object class that encodes the boolean primitive. This primitive has a unique encoding with the CER or DER rule, but the BER rule can support any byte value for the true value.

6.1. Predicate.

- asn-boolean-p

6.2. Inheritance.

- AsnNode

6.3. Constructors.

- *AsnBoolean → (none)*
  The *AsnBoolean* constructor creates a default asn boolean node.
- *AsnBoolean → (Boolean)*
  The *AsnBoolean* constructor creates an asn boolean node from a boolean object.

6.4. Methods.

- *to-boolean → Boolean (none)*
  The *to-boolean* method returns a *Boolean* object as the asn node representation.

7. Object AsnInteger

The *AsnInteger* class is the asn object class that encodes the integer primitive. This primitive has a unique encoding with the CER or DER rule. All encoding use a signed 2-complement form.

7.1. Predicate.

- asn-integer-p

7.2. Inheritance.

- AsnNode

7.3. Constructors.

- *AsnInteger → (none)*
  The *AsnInteger* constructor creates a default asn integer node.
- *AsnInteger → (Integer—Relatif)*
  The *AsnInteger* constructor creates an asn integer node from an integer or relatif object.

7.4. Methods.

- *to-relatif → Relatif (none)*
  The *to-relatif* method returns a *Relatif* object as the asn node representation.

8. Object AsnBits

The *AsnBits* class is the asn object class that encodes the bit string type. This type can be encoded either as a primitive or as constructed at sender's option. In CER form, the primitive form is used when the content length is less than 1000 octets, and the constructed form is used otherwise. The DER form will always use the primitive form.

8.1. Predicate.

- asn-bits-p

8.2. Inheritance.

- AsnNode

8.3. Constructors.

- *AsnBits → (none)*
  The *AsnBits* constructor creates a default asn bits node.
- *AsnBits → (String—Bitset)*
  The *AsnBits* constructor creates an asn bits node from a string or a bitset.

8.4. Methods.

- *to-bits → Bitset (none)*
  The *to-bits* method returns a *Bitset* object as a bit string representation.

9. Object AsnBmps

The *AsnBmps* class is the asn object class that encodes the asn bmp string primitive also known as the UCS-2 type string. This string is implemented, after conversion as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

9.1. Predicate.

- asn-bmps-p

9.2. Inheritance.

- AsnOctets

9.3. Constructors.

- *AsnBmps* → *(none)*
  The *AsnBmps* constructor creates a default asn string (BMP) node.
- *AsnBmps* → *(String)*
  The *AsnBmps* constructor creates an asn string (BMP) node from a string.

9.4. Methods.

- *to-string* → *String (none)*
  The *to-string* method returns a *String* object as a node representation.

10. Object AsnIas

The *AsnIas* class is the asn object class that encodes the IA5 string primitive. This string is implemented, after conversion as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

10.1. Predicate.

- asn-ias-p

10.2. Inheritance.

- AsnOctets

10.3. Constructors.

- *AsnIas* → *(none)*
  The *AsnIas* constructor creates a default asn string (IA5) node.
- *AsnIas* → *(String)*
  The *AsnIas* constructor creates an asn string (IA5) node from a string.

10.4. Methods.

- *to-string* → *String (none)*
  The *to-string* method returns a *String* object as a node representation.

11. Object AsnNums

The *AsnNums* class is the asn object class that encodes the asn numeric string primitive. This string is implemented, after conversion as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

11.1. Predicate.

- asn-nums-p

11.2. Inheritance.

- AsnOctets

11.3. Constructors.

- *AsnNums → (none)*
  The *AsnNums* constructor creates a default asn string (NUMERIC) node.
- *AsnNums → (String)*
  The *AsnNums* constructor creates an asn string (NUMERIC) node from a string.

11.4. Methods.

- *to-string → String (none)*
  The *to-string* method returns a *String* object as a node representation.

12. Object AsnPrts

The *AsnPrts* class is the asn object class that encodes the asn printable string primitive. This string is implemented, after conversion as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

12.1. Predicate.

- asn-prts-p

12.2. Inheritance.

- AsnOctets

12.3. Constructors.

- *AsnPrts* → *(none)*
  The *AsnPrts* constructor creates a default asn string (PRINTABLE) node.
- *AsnPrts* → *(String)*
  The *AsnPrts* constructor creates an asn string (PRINTABLE) node from a string.

12.4. Methods.

- *to-string* → *String (none)*
  The *to-string* method returns a *String* object as a node representation.

13. Object AsnUtfs

The *AsnUtfs* class is the asn object class that encodes the asn utf string primitive. This string is implemented as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

13.1. Predicate.

- asn-utfs-p

13.2. Inheritance.

- AsnOctets

13.3. Constructors.

- *AsnUtfs → (none)*
  The *AsnUtfs* constructor creates a default asn string (UNICODE) node.
- *AsnUtfs → (String)*
  The *AsnUtfs* constructor creates an asn string (UNICODE) node from a string.

13.4. Methods.

- *to-string → String (none)*
  The *to-string* method returns a *String* object as a node representation.

14. Object AsnUnvs

The *AsnUnvs* class is the asn object class that encodes the universal string primitive also known as the UCS-4 type string. This string is implemented, after conversion as an octet string. Consequently the rules for encoding in CER and DER modes are applied.

14.1. Predicate.

- asn-unvs-p

14.2. Inheritance.

- AsnOctets

14.3. Constructors.

- *AsnUnvs → (none)*
  The *AsnUnvs* constructor creates a default asn string (UNIVERSAL) node.
- *AsnUnvs → (String)*
  The *AsnUnvs* constructor creates an asn string (UNIVERSAL) node from a string.

14.4. Methods.

- *to-string → String (none)*
  The *to-string* method returns a *String* object as a node representation.

15. Object AsnGtm

The *AsnGtm* class is the asn object class that encodes the generalized time primitive. This primitive is encoded from its equivalent string representation. Although, the constructed mode is authorized, it does not make that much sense to use it.

15.1. Predicate.

- asn-gtm-p

15.2. Inheritance.

- AsnNode

15.3. Constructors.

- *AsnGtm → (none)*
  The *AsnGtm* constructor creates a default asn gtm node.
- *AsnGtm → (String)*
  The *AsnGtm* constructor creates an asn gtm node from a string.

15.4. Methods.

- *utc-p → Boolean (none)*
  The *utc-p* predicate returns true if the time is expressed in UTC mode.
- *to-time → Integer (none)*
  The *to-time* method returns a time representation of this asn node.
- *to-string → String (none)*
  The *to-string* method returns a string representation of this asn node.

16. Object AsnUtc

The *AsnUtc* class is the asn object class that encodes the utc time primitive. This primitive is encoding from its equivalent string representation. Although, the constructed mode is authorized, it does not make that much sense to use it.

16.1. Predicate.

- asn-utc-p

16.2. Inheritance.

- AsnNode

16.3. Constructors.

- *AsnUtc → (none)*
  The *AsnUtc* constructor creates a default asn utc node.
- *AsnUtc → (String)*
  The *AsnUtc* constructor creates an asn utc node from a string.

16.4. Methods.

- *utc-p → Boolean (none)*
  The *utc-p* predicate returns true if the time is expressed in UTC mode.
- *to-time → Integer (none)*
  The *to-time* method returns a time representation of this asn node.
- *to-string → String (none)*
  The *to-string* method returns a string representation of this asn node.

17. Object AsnSequence

The *AsnSequence* class is the asn object class that encodes the sequence constructed type. The order of elements is preserved in the encoding of the sequence.

17.1. Predicate.

- asn-sequence-p

17.2. Inheritance.

- AsnNode

17.3. Constructors.

- *AsnSequence → (none)*
  The *AsnSequence* constructor creates an empty asn sequence node.

17.4. Methods.

- *node-length → Integer (none)*
  The *node-length* method returns the number of nodes in the sequence.
- *node-add → none (AsnNode)*
  The *node-add* method adds a node to the sequence.
- *node-get → AsnNode (Integer)*
  The *node-get* method returns an asn node by index.

18. Object AsnSet

The *AsnSet* class is the asn object class that encodes the set constructed type. The order of elements is not important in a set.

18.1. Predicate.

- asn-set-p

18.2. Inheritance.

- AsnNode

18.3. Constructors.

- *AsnSet → (none)*
  The *AsnSet* constructor creates an empty asn set node.

18.4. Methods.

- *node-length → Integer (none)*
  The *node-length* method returns the number of nodes in the set.
- *node-add → none (AsnNode)*
  The *node-add* method adds a node to the set.
- *node-get → AsnNode (Integer)*
  The *node-get* method returns an asn node by index.

19. Object Oid

The *Oid* class is a base class that represents the X500 object identifier which is used in the ASN.1 encoding and in the X509 standard. An oid is simply represented by a vector of subidentifiers.

19.1. Predicate.

- oid-p

19.2. Inheritance.

- Object

19.3. Constructors.

- *Oid → (Integer—...)*
  The *Oid* constructor creates an oid from a sequence of integers.

19.4. Methods.

- *reset → none (none)*
  The *reset* method resets the oid object to its null empty state.
- *length → Integer (none)*
  The *length* method returns the length of the oid.
- *add → none (Integer—...)*
  The *add* method adds one or more sub-indentifiers to the oid.
- *get → Integer (Integer)*
  The *get* method returns an oid sub-identifier by index.
- *format → String (none)*
  The *format* method returns a string representation of the oid.

20. Object AsnOid

The *AsnOid* class is the asn object class that encodes the object identifier primitive. This primitive has a unique encoding with the CER or DER rule. The oid is built as a vector of subidentifiers (sid). Each sid is represented as an octa (64 bits) value.

20.1. Predicate.

- asn-oid-p

20.2. Inheritance.

- AsnNode

20.3. Constructors.

- *AsnOid → (Integer—...)*
  The *AsnOid* constructor creates an asn oid from a sequence of sid.

20.4. Methods.

- *sid-length → Integer (none)*
  The *length* method returns the length of the oid.
- *sid-add → none (Integer)*
  The *sid-add* method adds a sid the oid object.
- *sid-get → Integer (Integer)*
  The *sid-get* method returns a sid by oid index.
- *get-oid → Oid (none)*
  The *get-oid* method returns an oid object as the asn oid representation.

21. Object AsnRoid

The *AsnRoid* class is the asn object class that encodes the object relative identifier primitive. This primitive has a unique encoding with the CER or DER rule. The oid is built as a vector of subidentifiers (sid). Each sid is represented as an octa (64 bits) value. The difference with the oid object is to be found in the encoding of the first 2 sid.

21.1. Predicate.

- asn-roid-p

21.2. Inheritance.

- AsnNode

21.3. Constructors.

- *AsnRoid → (Integer—...)*
  The *AsnRoid* constructor creates an asn roid from a sequence of sid.

21.4. Methods.

- *sid-length → Integer (none)*
  The *length* method returns the length of the oid.
- *sid-add → none (Integer)*
  The *sid-add* method adds a sid the oid object.
- *sid-get → Integer (Integer)*
  The *sid-get* method returns a sid by oid index.
- *get-oid → Oid (none)*
  The *get-oid* method returns an oid object as the asn oid representation.

21.5. Functions.

- *asn-random-bits → none (Integer)*
  The *exit* function creates a random asn bit string. The argument is the number of bits in the random string.
- *asn-random-octets → none (Integer)*
  The *exit* function creates a random asn octet string. The integer argument is the number of octets in the string.

CHAPTER 5

Standard Math Module

The *Standard Mathematical* module is an original implementation of various mathematical facilities. The module can be divided into several catgeories which include convenient functions, linear algebra and real analysis.

1. Random number

The *math* module provides various functions that generate random numbers in different formats.

The numbers are generated with the help of the system random generator. Such generator is machine dependant and results can vary from one machine to another.

2. Primality testing

The *math* module provides various predicates that test a number for a primality condition. Most of these predicates are intricate and are normally not used except the *prime-probable-p* predicate.

The *fermat-p* and *miller-rabin-p* predicates return true if the primality condition is verified. These predicate operate with a base number. The prime number to test is the second argument.

2.1. Fermat primality testing. The *fermat-p* predicate is a simple primality test based on the "little Fermat theorem". A base number greater than 1 and less than the number to test must be given to run the test.

```
1 afnix:mth:fermat-p 2 7
```

In the preceeding example, the number 7 is tested, and the *fermat-p* predicate returns true. If a number is prime, it is guaranted to pass the test. The opposite is not true. For example, 561 is a composite number, but the Fermat test will succeed with the base

Function	Description
get-random-integer	return a random integer number
get-random-real	return a random real number between 0.0 and 1.0
get-random-relatif	return a random relatif number
get-random-prime	return a random probable prime relatif number

Predicate	Description
fermat-p	Fermat test predicate
miller-rabin-p	Miller-Rabin test predicate
prime-probable-p	general purpose prime probable test
get-random-prime	return a random probable prime relatif number

Operator	Description
==	compare two vectors for equality
!=	compare two vectors for difference
?=	compare two vectors upto a precision
+=	add a scalar or vector to the vector
-=	substract a scalar or vector to the vector
*=	multiply a scalar or vector to the vector
/=	divide a vector by a scalar

Method	Description
set	set a vector component by index
get	get a vector component by index
clear	clear a vector
reset	reset a vector
get-size	get the vector dimension
dot	compute the dot product with another vector
norm	compute the vector norm

2. Numbers that successfully pass the Fermat test but which are composite are called Carmichael numbers. For those numbers, a better test needs to be employed, such like the Miller-Rabin test.

2.2. Miller-Rabin primality testing. The *miller-rabin-p* predicate is a complex primality test that is more efficient in detecting prime number at the cost of a longer computation. A base number greater than 1 and less than the number to test must be given to run the test.

```
1 afnix:mth:miller-rabin-p 2 561
```

In the preceeding example, the number 561, which is a Carmichael number, is tested, and the *miller-rabin-p* predicate returns false. The probability that a number is prime depends on the number of times the test is ran. Numerous studies have been made to determine the optimal number of passes that are needed to declare that a number is prime with a good probability. The *prime-probable-p* predicate takes care to run the optimal number of passes.

2.3. General primality testing. The *prime-probable-p* predicate is a complex primality test that incorporates various primality tests. To make the story short, the prime candidate is first tested with a series of small prime numbers. Then a fast Fermat test is executed. Finally, a series of Miller-Rabin tests are executed. Unlike the other primality tests, this predicate operates with a number only and optionally, the number of test passes. This predicate is the recommended test for the folks who want to test their numbers.

```
1 afnix:mth:prime-probable-p 17863
```

3. Linear algebra

The *math* module provides an original and extensive support for linear and non linear algebra. This includes vector, matrix and solvers. Complex methods for non linear operations are also integrated tightly in this module.

3.1. Real vector. The *math* module provides the *Rvector* object which implements the real vector interface *Rvi* . Such interface provides numerous operators and methods for manipulating vectors as traditionnaly found in linear algebra packages.

Operator	Description
==	compare two matrices for equality
!=	compare two matrices for difference
?=	compare two matrices upto a precision

Method	Description
set	set a matrix component by index
get	get a matrix component by index
clear	clear a vector
get-row-size	get the matrix row dimension
get-col-size	get the matrix column dimension
norm	compute the matrix norm

3.2. Creating a vector. A vector is always created by size. A null size is perfectly valid. When a vector is created, it can be filled by setting the components by index.

```
1 # create a simple vector
2 const rv (afnix:mth:Rvector 3)
3 # set the components by index
4 rv:set 0 0.0
5 rv:set 1 3.0
6 rv:set 2 4.0
```

3.3. Real matrix. The *math* module provides the *Rmatrix* object which implements the real matrix interface *Rmi* . This interface is designed to operate with the vector interface and can handle sparse or full matrix.

CHAPTER 6

Standard Math Reference

1. Object Rvi

The *Rvi* class an abstract class that models the behavior of a real based vector. The class defines the vector length as well as the accessor and mutator methods.

1.1. Predicate.

- rvi-p

1.2. Inheritance.

- Serial

1.3. Operators.

- *== → Boolean (Vector)*
  The == operator returns true if the calling object is equal to the vector argument.
- *!= → Boolean (Vector)*
  The == operator returns true if the calling object is not equal to the vector argument.
- *?= → Boolean (Vector)*
  The *?=* operator returns true if the calling object is equal to the vector argument upto a certain precision.
- *+= → Vector (Real—Vector)*
  The *+=* operator returns the calling vector by adding the argument object. In the first form, the real argument is added to all vector components. In the second form, the vector components are added one by one.
- *-= → Vector (Real—Vector)*
  The -= operator returns the calling vector by substracting the argument object. In the first form, the real argument is substracted to all vector components. In the second form, the vector components are substracted one by one.
- **= → Vector (Real—Vector)*
  The **=* operator returns the calling vector by multiplying the argument object. In the first form, the real argument is multiplied to all vector components. In the second form, the vector components are multiplied one by one.
- */= → Vector (Real)*
  The */=* operator returns the calling vector by dividing the argument object. The vector components are divided by the real argument.

1.4. Methods.

- *set → None (Integer Real)*
  The *set* method sets a vector component by index.
- *get → Real (Integer)*
  The *get* method gets a vector component by index.
- *clear → None (None)*
  The *clear* method clears a vector. The dimension is not changed.
- *reset → None (None)*
  The *reset* method resets a vector. The size is set to 0.
- *get-size → Real (None)*
  The *get-size* method returns the vector dimension.
- *dot → Real (Vector)*
  The *dot* method computes the dot product with the vector argument.
- *norm → Real (None)*
  The *norm* method computes the vector norm.

- *permutate → Vector (Cpi)*
  The *permutate* method permutates the vector components with the help of a combinatoric permutation object.
- *reverse → Vector (Cpi)*
  The *reverse* method reverse (permutate) the vector components with the help of a combinatoric permutation object.

2. Object Rvector

The *Rvector* class is the default implementation of the real vector interface.

2.1. Predicate.

- r-vector-p

2.2. Inheritance.

- Rvi

2.3. Constructors.

- *Rvector → (None)*
  The *Rvector* constructor creates a default null real vector.
- *Rvector → (Integer)*
  The *Rvector* constructor creates a real vector those dimension is given as the calling argument.

2.4. Functions.

- *get-random-integer → Integer (none—Integer)*
  The *get-random-integer* function returns a random integer number. Without argument, the integer range is machine dependent. With one integer argument, the resulting integer number is less than the specified maximum bound.
- *get-random-real → Real (none—Boolean)*
  The *get-random-real* function returns a random real number between 0.0 and 1.0. In the first form, without argument, the random number is between 0.0 and 1.0 with 1.0 included. In the second form, the boolean flag controls whether or not the 1.0 is included in the result. If the argument is false, the 1.0 value is guaranted to be excluded from the result. If the argument is true, the 1.0 is a possible random real value. Calling this function with the argument set to true is equivalent to the first form without argument.
- *get-random-relatif → Relatif (Integer—Integer Boolean)*
  The *get-random-relatif* function returns a n bits random positive relatif number. In the first form, the argument is the number of bits. In the second form, the first argument is the number of bits and the second argument, when true produce an odd number, or an even number when false.
- *get-random-prime → Relatif (Integer)*
  The *get-random-prime* function returns a n bits random positive relatif probable prime number. The argument is the number of bits. The prime number is generated by using the Miller-Rabin primality test. As such, the returned number is declared probable prime. The more bits needed, the longer it takes to generate such number.
- *get-random-bitset → Bitset (Integer)*
  The *get-random-bitset* function returns a n bits random bitset. The argument is the number of bits.
- *fermat-p → Boolean (Integer—Relatif Integer—Relatif)*
  The *fermat-p* predicate returns true if the little fermat theorem is validated. The first argument is the base number and the second argument is the prime number to validate.
- *miller-rabin-p → Boolean (Integer—Relatif Integer—Relatif)*
  The *miller-rabin-p* predicate returns true if the Miller-Rabin test is validated. The first argument is the base number and the second argument is the prime number to validate.

- *prime-probable-p* → *Boolean (Integer—Relatif [Integer])*
  The *prime-probable-p* predicate returns true if the argument is a probable prime. In the first form, only an integer or relatif number is required. In the second form, the number of iterations is specified as the second argument. By default, the number of iterations is specified to 56.

CHAPTER 7

Standard Networking Module

The *Standard Networking* module is an original implementation of networking facilities for the Internet Protocol. The module features standard TCP and UDP sockets for point to point communication as well as multicast socket. Numerous functions and objects for address manipulation are also included in this module. This module is also designed to support IP version 6 with certain platforms.

1. IP address

The IP based communication uses a standard address to reference a particular peer. With IP version 4, the standard dot notation is with 4 bytes. With IP version 6, the standard semicolon notation is with 16 bytes. The current implementation supports both versions.

```
127.0.0.1 # ipv4 localhost
0:0:0:0:0:0:0:1 # ipv6 localhost
```

IP address architecture and behavior are described in various documents as listed in the bibliography.

1.1. Domain name system. The translation between a host name and an IP address is performed by a *resolver* which uses the *Domain Name System* or DNS. Access to the DNS is automatic with the implementation. Depending on the machine resolver configuration, a particular domain name translation might result in an IP version 4 or IP version 6 address. Most of the time, an IP version 4 address is returned.

The mapping between an IP address and a host name returns the associated *canonical name* for that IP address. This is the reverse of the preceding operation.

2. The Address class

The *Address* class allows manipulation of IP address. The constructor takes a string as its arguments. The argument string can be either an IP address or a host name which can be qualified or not. When the address is constructed with a host name, the IP address resolution is done immediately.

2.1. Name to address translation. The most common operation is to translate a host name to its equivalent IP address. Once the *Address* object is constructed, the *get-address* method returns a string representation of the internal IP address. The following example prints the IP address of the localhost, that is *127.0.0.1* with IP version 4.

```
# load network module
interp:library "afnix-net"
# get the localhost address
const addr (afnix:net:Address "localhost")
# print the ip address
println (addr:get-address)
```

As another example, the *get-host-name* function returns the host name of the running machine. The previous example can be used to query its IP address.

2.2. Address to name translation. The reverse operation of name translation maps an IP address to a *canonical name* . It shall be noted that the reverse lookup is not done automatically, unless the *reverse flag* is set in the constructoor. The *get-canonical-name* method of the *Address* class returns such name. Example `XNET001.als` is a demonstration program which prints the address original name, the IP address and the canonical name. Fell free to use it with your favorite site to check the equivalence between the original name and the canonical name.

```
# print the ip address information of the arguments
# usage: axi XNET001.als [hosts ...]
# get the network module
interp:library "afnix-net"
# print the ip address
const ip-address-info (host) {
  try {
    const addr (afnix:net:Address host true)
    println "host name : " (addr:get-name)
    println " ip address : " (addr:get-address)
    println " canonical name : " (
      addr:get-canonical-name)
    # get aliases
    const size (addr:get-alias-size)
    loop (trans i 0) (< i size) (i:++) {
      println " alias address : " (
        addr:get-alias-address i)
      println " alias name : " (
        addr:get-alias-name i)
    }
  } (errorln "error: " what:reason)
}
# get the hosts
for (s) (interp:argv) (ip-address-info s)
zsh> axi net-0001.als localhost
host name : localhost
ip address : 127.0.0.1
canonical name : localhost
```

2.3. Address operations. The *Address* class provides several methods and operators that ease the address manipulation in a protocol indepedant way. For example, the == operator compares two addresses. The ++ operator can also be used to get the next IP address.

3. Transport layers

The two transport layer protocols supported by the Internet protocol is the TCP, a full-duplex oriented protocol, and UDP, a datagram protocol. TCP is a reliable protocol while UDP is not. By reliable, we mean that the protocol provides automatically some mechanisms for error recovery, message delivery, acknowledgment of reception, etc... The use of TCP vs. UDP is dictated mostly by the reliability concerns, while UDP reduces the traffic congestion.

3.1. Service port. In the client-server model, a connection is established between two hosts. The connections is made via the IP address and the port number. For a given service, a port identifies that service at a particular address. This means that multiple services can exist at the same address. More precisely, the transport layer protocol is also used to distinguish a particular service. The network module provides a simple mechanism to retrieve the port number, given its name and protocol. The function *get-tcp-service* and *get-udp-service* returns the port number for a given service by name. For example, the *daytime* server is located at port number 13.

```
assert 13 (afnix:net:get-tcp-service "daytime")
assert 13 (afnix:net:get-udp-service "daytime")
```

3.2. Host and peer. With the client server model, the only information needed to identify a particular client or server is the address and the port number. When a client connects to a server, it specify the port number the server is operating. The client uses a random port number for itself. When a server is created, the port number is used to bind the server to that particular port. If the port is already in use, that binding will fail. From a reporting point of view, a connection is therefore identified by the running host address and port, and the peer address and port. For a client, the peer is the server. For a server, the peer is the client.

4. TCP client socket

The *TcpClient* class creates an TCP client object by address and port. The address can be either a string or an *Address* object. During the object construction, the connection is established with the server. Once the connection is established, the client can use the *read* and *write* method to communicate with the server. The *TcpClient* class is derived from the *Socket* class which is derived from the *InputStream* and *OutputStream* classes.

4.1. Day time client. The simplest example is a client socket which communicates with the daytime server. The server is normally running on all machines and is located at port 13.

```
# get the network module
interp:library "afnix-net"
# get the daytime server port
const port (afnix:net:get-tcp-service "daytime")
# create a tcp client socket
const s (afnix:net:TcpClient "localhost" port)
# read the data - the server close the connection
while (s:valid-p) (println (s:readln))
```

Example `3201.als` in the example directory prints the day time of the local host without argument or the day time of the argument. Feel free to use it with *www.afnix.org* . If the server you are trying to contact does not have a day time server, an exception will be raised and the program terminates.

```
zsh> axi 3201.als www.afnix.org
```

4.2. HTTP request example. Another example which illustrates the use of the *TcpClient* object is a simple client which download a web page. At this stage we are not concern with the URL but rather the mechanics involved. The request is made by opening a TCP client socket on port 80 (the HTTP server port) and sending a request by writing some HTTP commands. When the commands have been sent, the data sent by the server are read and printed on the standard output. Note that this example is not concerned by error detection.

```
# fetch an html page by host and page
# usage: axi 3203.als [host] [page]
# get the network module
interp:library "afnix-net"
interp:library "afnix-sys"
# connect to the http server and issue a request
const send-http-request (host page) {
  # create a client sock on port 80
  const s (afnix:net:TcpClient host 80)
  const saddr (s:get-socket-address)
  # format the request
```

```
  s:writeln "GET " page " HTTP/1.1"
  s:writeln "Host: " (saddr:get-canonical-name)
  s:writeln "Connection: close"
  s:writeln "User-Agent: afnix tcp client example"
  s:newline
  # write the result
  while (s:valid-p) (println (s:readln))
}
# get the argument
if (!= (interp:argv:length) 2) (afnix:sys:exit 1)
const host (interp:argv:get 0)
const page (interp:argv:get 1)
# send request
send-http-request host page
```

5. UDP client socket

UDP client socket is similar to TCP client socket. However, due to the unreliable nature of UDP, UDP clients are somehow more difficult to manage. Since there is no flow control, it becomes more difficult to assess whether or not a datagram has reached its destination. The same apply for a server, where a reply datagram might be lost. The *UdpClient* class is the class which creates a UDP client object. Its usage is similar to the *TcpClient* .

5.1. The time client. The UDP time server normally runs on port 37 is the best place to enable it. A UDP client is created with the *UdpClient* class. Once the object is created, the client sends an empty datagram to the server. The server send a reply datagram with 4 bytes, in network byte order, corresponding to the date as of January 1st 1900. Example `3204.als` prints date information after contacting the local host time server or the host specified as the first argument.

```
# get the libraries
interp:library "afnix-net"
interp:library "afnix-sys"
# get the daytime server port
const port (afnix:net:get-udp-service "time")
# create a client socket and read the data
const print-time (host) {
  # create a udp client socket
  const s (afnix:net:UdpClient host port)
  # send an empty datagram
  s:write
  # read the 4 bytes data and adjust to epoch
  const buf (s:read 4)
  const val (- (buf:get-quad) 2208988800)
  # format the date
  const time (afnix:sys:Time val)
  println (time:format-date) ' ' (time:format-time)
}
# check for one argument or use localhost
const host (if (== (interp:argv:length) 0)
  "localhost" (interp:argv:get 0))
print-time host
```

This example calls for several comments. First the *write* method without argument sends an empty datagram. It is the datagram which trigger the server. The *read* method reads 4 bytes from the reply datagram and places them in a *Buffer* object. Since the bytes are in network byte order, the conversion into an integer value is done with the *get-quad* method. Finally, in order to use the *Time* class those epoch is January 1st 1970, the constant *2208988800* is subtracted from the result. Remember that the time server sends the date in reference to January 1st 1900. More information about the time server can be found in RFC738.

5.2. More on reliability. The previous example has some inherent problems due to the unreliability of UDP. If the first datagram is lost, the *read* method will block indefinitely. Another scenario which causes the *read* method to block is the loss of the server reply datagram. Both problem can generally be fixed by checking the socket with a timeout using the *valid-p* method. With one argument, the method timeout and return false. In this case, a new datagram can be send to the server. Example `3205.als` illustrates this point. We print below the extract of code.

```
# create a client socket and read the data
const print-time (host) {
  # create a udp client socket
  const s (afnix:net:UdpClient host port)
  # send an empty datagram until the socket is valid
  s:write
  # retransmit datagram each second
  while (not (s:valid-p 1000)) (s:write)
  # read the 4 bytes data and adjust to epoch
  const buf (s:read 4)
  const val (- (buf:get-quad) 2208988800)
  # format the date
  const time (afnix:sys:Time val)
  println (time:format-date) ' ' (time:format-time)
}
```

Note that this solution is a naive one. In the case of multiple datagrams, a sequence number must be placed because there is no clue about the lost datagram. A simple rule of thumb is to use TCP as soon as reliability is a concern, but this choice might not so easy.

5.3. Error detection. Since UDP is not reliable, there is no simple solution to detect when a datagram has been lost. Even worse, if the server is not running, it is not easy to detect that the client datagram has been lost. In such situation, the client might indefinitely send datagram without getting an answer. One solution to this problem is again to count the number of datagram re-transmit and eventually give up after a certain time.

6. Socket class

The *Socket* class is the base class for both *TcpClient* and *UdpClient* . The class provides methods to query the socket port and address as well as the peer port and address. Note at this point that the UDP socket is a connected socket. Therefore, these methods will work fine. The *get-socket-address* and *get-socket-port* returns respectively the address and port of the connected socket. The *get-peer-address* and *get-peer-port* returns respectively the address and port of the connected socket's peer. Example `3206.als` illustrates the use of these methods.

```
# create a client socket and read the data
const print-socket-info (host) {
  # create a tcp client socket
  const s (afnix:net:TcpClient host port)
  # print socket address and port
  const saddr (s:get-socket-address)
  const sport (s:get-socket-port)
  println "socket ip address : " (
    saddr:get-address)
  println "socket canonical name : " (
    saddr:get-canonical-name)
  println "socket port : " sport
  # print peer address and port
  const paddr (s:get-peer-address)
  const pport (s:get-peer-port)
  println "peer ip address : " (
    paddr:get-address)
  println "peer canonical name : " (
```

```
    paddr:get-canonical-name)
  println "peer port : " pport
}
```

6.1. Socket predicates. The *Socket* class is associated with the *socket-p* predicate. The respective client objects have the *tcp-client-p* predicate and *udp-client-p* predicate.

7. TCP server socket

The *TcpServer* class creates an TCP server object. There are several constructors for the TCP server. In its simplest form, without port, a TCP server is created on the *localhost* with an ephemeral port number (i.e port 0 during the call). With a port number, the TCP server is created on the *localhost* . For a multi-homed host, the address to use to run the server can be specified as the first argument. The address can be either a string or an *Address* object. In both cases, the port is specified as the second argument. Finally, a third argument called the *backlog* can be specified to set the number of acceptable incoming connection. That is the maximum number of pending connection while processing a connection. The following example shows various ways to create a TCP server.

```
trans s (afnix:net:TcpServer)
trans s (afnix:net:TcpServer 8000)
trans s (afnix:net:TcpServer 8000 5)
trans s (afnix:net:TcpServer "localhost" 8000)
trans s (afnix:net:TcpServer "localhost" 8000 5)
trans s (afnix:net:TcpServer (
    Address "localhost") 8000)
trans s (afnix:net:TcpServer (
    Address "localhost") 8000 5)
```

7.1. Echo server example. A simple *echo server* can be built and tested with the standard `telnet` application. The application will echo all lines that are typed with the `telnet` client. The server is bound on the port 8000, since ports 0 to 1024 are privileged ports.

```
# get the network module
interp:library "afnix-net"
# create a tcp server on port 8000
const srv (afnix:net:TcpServer 8000)
# wait for a connection
const s (srv:accept)
# echo the line until the end
while (s:valid-p) (s:writeln (s:readln))
```

The `telnet` session is then quite simple. The line *hello world* is echoed by the server.

```
zsh> telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello world
^D
```

7.2. The accept method. The previous example illustrates the mechanics of a server. When the server is created, the server is ready to accept connection. The *accept* method blocks until a client connect with the server. When the connection is established, the *accept* method returns a socket object which can be used to read and write data.

7.3. Multiple connections. One problem with the previous example is that the server accepts only one connection. In order to accept multiple connection, the *accept* method must be placed in a loop, and the server operation in a thread (There are some situations where a new process might be more appropriate than a thread). Example `3302.als` illustrates such point.

```
# get the network module
interp:library "afnix-net"
# this function echo a line from the client
const echo-server (s) {
  while (s:valid-p) (s:writeln (s:readln))
}
# create a tcp server on port 8000
const srv (afnix:net:TcpServer 8000)
# wait for a connection
while true {
  trans s (srv:accept)
  launch (echo-server s)
}
```

8. UDP server socket

The *UdpServer* class is similar to the *TcpServer* object, except that there is no backlog parameters. In its simplest form, the UDP server is created on the *localhost* with an ephemeral port (i.e port 0). With a port number, the server is created on the *localhost* . For a multi-homed host, the address used to run the server can be specified as the first argument. The address can be either a string or an *Address* object. In both cases, the port is specified as the second argument.

```
trans s (afnix:net:UdpServer)
trans s (afnix:net:UdpServer 8000)
trans s (afnix:net:UdpServer "localhost" 8000)
trans s (afnix:net:UdpServer (
    Address "localhost") 8000)
```

8.1. Echo server example. The *echo server* can be revisited to work with udp datagram. The only difference is the use of the *accept* method. For a UDP server, the method return a *Datagram* object which can be used to read and write data.

```
# get the network module
interp:library "afnix-net"
# create a udp server on port 8000
const srv (afnix:net:UdpServer 8000)
# wait for a connection
while true {
  trans dg (srv:accept)
  dg:writeln (dg:readln)
}
```

8.2. Datagram object. With a UDP server, the *accept* method returns a *Datagram* object. Because a UDP is connection-less, the server has no idea from whom the datagram is coming until that one has been received. When a datagram arrives, the *Datagram* object is constructed with the peer address being the source address. Standard i/o methods can be used to read or write. When a write method is used, the data are sent back to the peer in a form of another datagram.

```
# wait for a datagram
trans dg (s:accept)
# assert datagram type
assert true (datagram-p dg)
# get contents length
```

```
println "datagram buffer size : " (dg:get-buffer-length)
# read a line from this datagram
trans line (dg:readln)
# send it back to the sender
s:writeln line
```

8.3. Input data buffer. For a datagram, and generally speaking, for a UDP socket, all input operations are buffered. This means that when a datagram is received, the *accept* method places all data in an input buffer. This means that a read operation does not necessarily flush the whole buffer but rather consumes only the requested character. For example, if one datagram contains the string *hello world* . A call to *readln* will return the entire string. A call to read will return only the character 'h'. Subsequent call will return the next available characters. A call like *read 5* will return a buffer with 5 characters. Subsequent calls will return the remaining string. In any case, the *get-buffer-length* will return the number of available characters in the buffer. A call to *valid-p* will return true if there are some characters in the buffer or if a new datagram has arrived. Care should be taken with the *read* method. For example if there is only 4 characters in the input buffer and a call to read for 10 characters is made, the method will block until a new datagram is received which can fill the remaining 6 characters. Such situation can be avoided by using the *get-buffer-length* and the *valid-p* methods. Note also that a timeout can be specified with the *valid-p* method.

9. Low level socket methods

Some folks always prefer to do everything by themselves. Most of the time for good reasons. If this is your case, you might have to use the low level socket methods. Instead of using a client or server class, the implementation let's you create a *TcpSocket* or *UdpSocket* . Once this done, the *bind* , *connect* and other methods can be used to create the desired connection.

9.1. A socket client. A simple TCP socket client is created with the *TcpSocket* class. Then the *connect* method is called to establish the connection.

```
# create an address and a tcp socket
const addr (afnix:net:Address "localhost")
const sid (afnix:net:TcpSocket)
# connect the socket
sid:connect 13 addr
```

Once the socket is connected, normal read and write operations can be performed. After the socket is created, it is possible to set some options. A typical one is *NO-DELAY* which disable the Naggle algorithm.

```
# create an address and a tcp socket
const addr (afnix:net:Address "localhost")
const sid (afnix:net:TcpSocket)
# disable the naggle algorithm
sid:set-option sid:NO-DELAY true
# connect the socket
sid:connect 13 addr
```

CHAPTER 8

Networking reference

1. Object Address

The *Address* class is the Internet address manipulation class. The class can be used to perform the conversion between a host name and an IP address. The opposite is also possible. Finally, the class supports both IP version 4 and IP version 6 address formats.

1.1. Predicate.

- address-p

1.2. Inheritance.

- Object

1.3. Constructors.

- *Address → (String)*
  The *Address* constructor create an IP address object by name. The name argument is a string of a host name or a valid IP address representation.
- *Address → (String Boolean)*
  The *Address* constructor create an IP address object by name and force the reverse lookup resolution depending on the boolean flag value. The first argument is a string of a host name or a valid IP address representation. The second argument is a boolean flag that indicates whether or not reverse lookup must occur during the construction.

1.4. Operators.

- *== → Boolean (Address)*
  The == operator returns true if the calling object is equal to the address argument.
- *!= → Boolean (Address)*
  The *!=* operator returns true if the calling object is not equal to the address argument.
- *< → Boolean (Address)*
  The < operator returns true if the calling address is less than the address object.
- *<= → Boolean (Address)*
  The <= operator returns true if the calling address is less equal than the address object.
- *> → Boolean (Address)*
  The > operator returns true if the calling address is greater than the address object.
- *>= → Boolean (Address)*
  The <= operator returns true if the calling address is greater equal than the address object.
- *++ → Address (Address)*
  The *++* operator increments the calling address by one position.

1.5. Methods.

- *resolve → String Boolean (none)*
  The *resolve* method resolves an host name and eventually performs a reverse lookup. The first argument is a string of a host name or a valid IP address representation. The second argument is a boolean flag that indicates whether or not reverse lookup must occur during the resolution.
- *get-name → String (none)*
  The *get-name* method returns the original name used during the object construction.

- *get-address → String (none)*
  The *get-address* method returns a string representation of the IP address. The string representation follows the IP version 4 or IP version 6 preferred formats, depending on the internal representation.
- *get-vector → Vector (none)*
  The *get-vector* method returns a vector representation of the IP address. The vector result follows the IP version 4 or IP version 6 preferred format, depending on the internal representation.
- *get-canonical-name → String (none)*
  The *get-canonical-name* method returns a fully qualified name of the address. The resulting name is obtained by performing a reverse lookup. Note that the name can be different from the original name.
- *get-alias-size → Integer (none)*
  The *get-alias-size* method returns the number of aliases for the address. The number of aliases includes as well the primary resolved name which is located at index 0.
- *get-alias-name → String (Integer)*
  The *get-alias-name* method returns a fully qualified name of the address alias by index. The first argument is the alias index number which must be in the alias index range. The resulting name is obtained by performing a reverse lookup. Note that the name can be different from the original name. Using index 0 is equivalent to call *get-canonical-name* .
- *get-alias-address → String (Integer)*
  The *get-alias-address* method returns a string representation of the IP address alias by index. The first argument is the alias index number which must be in the alias index range. The string representation follows the IP version 4 or IP version 6 preferred formats, depending on the internal representation. Using index 0 is equivalent to call *get-address* .
- *get-alias-vector → Vector (Integer)*
  The *get-alias-vector* method returns a vector representation of the IP address alias by index. The first argument is the alias index number which must be in the alias index range. The vector result follows the IP version 4 or IP version 6 preferred format, depending on the internal representation. Using index 0 is equivalent to call *get-vector* .

1.6. Functions.

- *get-loopback → String (none)*
  The *get-loopback* function returns the name of the machine loopback. On a UNIX system, that name is `localhost` .
- *get-tcp-service → String (Integer)*
  The *get-tcp-service* function returns the name of the tcp service given its port number. For example, the tcp service at port 13 is the `daytime` server.
- *get-udp-service → String (Integer)*
  The *get-udp-service* function returns the name of the udp service given its port number. For example, the udp service at port 19 is the `chargen` server.

2. Object Socket

The *Socket* class is a base class for the **AFNIX** network services. The class is automatically constructed by a derived class and provide some common methods for all socket objects.

2.1. Predicate.

- socket-p

2.2. Inheritance.

- InputStreamOutputStream

2.3. Constants.

- *REUSE-ADDRESS → ()*
  The *REUSE-ADDRESS* constant is used by the *set-option* method to enable socket address reuse. This option changes the rules that validates the address used by bind. It is not recommended to use that option as it decreases TCP reliability.
- *BROADCAST → ()*
  The *BROADCAST* constant is used by the *set-option* method to enable broadcast of packets. This options only works with IP version 4 address. The argument is a boolean flag only.
- *DONT-ROUTE → ()*
  The *DONT-ROUTE* constant is used by the *set-option* method to control if a packet is to be sent via the routing table. This option is rarely used with **AFNIX** . The argument is a boolean flag only.
- *KEEP-ALIVE → ()*
  The *KEEP-ALIVE* constant is used by the *set-option* method to check periodically if the connection is still alive. This option is rarely used with **AFNIX** . The argument is a boolean flag only.
- *LINGER → ()*
  The *LINGER* constant is used by the *set-option* method to turn on or off the lingering on close. If the first argument is true, the second argument is the linger time.
- *RCV-SIZE → ()*
  The *RCV-SIZE* constant is used by the *set-option* method to set the receive buffer size.
- *SND-SIZE → ()*
  The *SND-SIZE* constant is used by the *set-option* method to set the send buffer size.
- *HOP-LIMIT → ()*
  The *HOP-LIMIT* constant is used by the *set-option* method to set packet hop limit.
- *MULTICAST-LOOPBACK → ()*
  The *MULTICAST-LOOPBACK* constant is used by the *set-option* method to control whether or not multicast packets are copied to the loopback. The argument is a boolean flag only.
- *MULTICAST-HOP-LIMIT → ()*
  The *MULTICAST-HOP-LIMIT* constant is used by the *set-option* method to set the hop limit for multicast packets.
- *MAX-SEGMENT-SIZE → ()*
  The *MAX-SEGMENT-SIZE* constant is used by the *set-option* method to set the TCP maximum segment size.

- *NO-DELAY → ()*
  The *NO-DELAY* constant is used by the *set-option* method to enable or disable the Naggle algorithm.

2.4. Methods.

- *bind → none (Integer)*
  The *bind* method binds this socket to the port specified as the argument.
- *bind → none (Integer Address)*
  The *bind* method binds this socket to the port specified as the first argument and the address specified as the second argument.
- *connect → none (Integer Address [Boolean])*
  The *connect* method connects this socket to the port specified as the first argument and the address specified as the second argument. A connected socket is useful with udp client that talks only with one fixed server. The optional third argument is a boolean flag that permits to select whether or not the alias addressing scheme should be used. If the flag is false, the default address is used. If the flag is true, an attempt is made to connect to the first successful address that is part of the alias list.
- *open-p → Boolean (none)*
  The *open-p* predicate returns true if the socket is open. The method checks that a descriptor is attached to the object. This does not mean that the descriptor is valid in the sense that one can read or write on it. This method is useful to check if a socket has not been closed.
- *shutdown → Boolean (none—Boolean)*
  The *shutdown* method shutdowns or close the connection. Without argument, the connection is closed without consideration for those symbols attached to the object. With one argument, the connection is closed in one direction only. If the mode argument is false, further receive is disallowed. If the mode argument is true, further send is disallowed. The method returns true on success, false otherwise.
- *ipv6-p → Boolean (none)*
  The *ipv6-p* predicate returns true if the socket address is an IP version 6 address, false otherwise.
- *get-socket-address → Address (none)*
  The *get-socket-address* method returns an address object of the socket. The returned object can be later used to query the canonical name and the ip address.

- *get-socket-port → Integer (none)*
  The *get-socket-port* method returns the port number of the socket.
- *get-socket-authority → String (none)*
  The *get-socket-authority* method returns the authority string in the form of an address and port pair of the socket.
- *get-peer-address → Address (none)*
  The *get-peer-address* method returns an address object of the socket's peer. The returned object can be later used to query the canonical name and the ip address.

- *get-peer-port → Integer (none)*
  The *get-peer-port* method returns the port number of the socket's peer.
- *get-peer-authority → String (none)*
  The *get-peer-authority* method returns the authority string in the form of an address and port pair of the socket's peer.

- *set-option → Boolean (constant [Boolean—Integer] [Integer])*
  The *set-option* method set a socket option. The first argument is the option to set. The second argument is a boolean value which turn on or off the option. The optional third argument is an integer needed for some options.
- *set-encoding-mode → none (Item—String)*
  The *set-encoding-mode* method sets the input and output encoding mode. In the first form, with an item, the stream encoding mode is set directly. In the second form, the encoding mode is set with a string and might also alter the stream transcoing mode.
- *set-input-encoding-mode → none (Item—String)*
  The *set-input-encoding-mode* method sets the input encoding mode. In the first form, with an item, the stream encoding mode is set directly. In the second form, the encoding mode is set with a string and might also alter the stream transcoing mode.
- *get-input-encoding-mode → Item (none)*
  The *get-input-encoding-mode* method return the input encoding mode.
- *set-output-encoding-mode → none (Item—String)*
  The *set-output-encoding-mode* method sets the output encoding mode. In the first form, with an item, the stream encoding mode is set directly. In the second form, the encoding mode is set with a string and might also alter the stream transcoing mode.
- *get-output-encoding-mode → Item (none)*
  The *get-output-encoding-mode* method return the output encoding mode.

3. Object TcpSocket

The *TcpSocket* class is a base class for all tcp socket objects. The class is derived from the *Socket* class and provides some specific tcp methods. If a *TcpSocket* is created, the user is responsible to connect it to the proper address and port.

3.1. Predicate.

- tcp-socket-p

3.2. Inheritance.

- Socket

3.3. Constructors.

- *TcpSocket → (none)*
  The *TcpSocket* constructor creates a new tcp socket.

3.4. Methods.

- *accept → TcpSocket (none)*
  The *accept* method waits for incoming connection and returns a *TcpSocket* object initialized with the connected peer. The result socket can be used to perform i/o operations. This method is used by tcp server.
- *listen → Boolean (none—Integer)*
  The *listen* method initialize a socket to accept incoming connection. Without argument, the default number of incoming connection is 5. The integer argument can be used to specify the number of incoming connection that socket is willing to queue. This method is used by tcp server.

4. Object TcpClient

The *TcpClient* class creates a tcp client by host and port. The host argument can be either a name or an address object. The port argument is the server port to contact. The *TcpClient* class is derived from the *TcpSocket* class. This class has no specific methods.

4.1. Predicate.

- tcp-client-p

4.2. Inheritance.

- TcpSocket

4.3. Constructors.

- *TcpClient → (String Integer)*
  The *TcpClient* constructor creates a new tcp client socket by host name and port number.

5. Object TcpServer

The *TcpServer* class creates a tcp server by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The *TcpServer* class is derived from the *TcpSocket* class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The backlog can be specified as the last argument. The host name can also be specified as the first argument, the port as second argument and eventually the backlog. Note that the host can be either a string or an address object.

5.1. Predicate.

- tcp-server-p

5.2. Inheritance.

- TcpSocket

5.3. Constructors.

- *TcpServer → (none)*
  The *TcpServer* constructor creates a default tcp server.
- *TcpServer → (Integer)*
  The *TcpServer* constructor creates a default tcp server which is bound on the specified port argument.
- *TcpServer → (Integer Integer)*
  The *TcpServer* constructor creates a default tcp server which is bound on the specified port argument. The second argument is the backlog value.
- *TcpServer → (String Integer)*
  The *TcpServer* constructor creates a tcp server by host name and port number. The first argument is the host name. The second argument is the port number.
- *TcpServer → (String Integer Integer)*
  The *TcpServer* constructor creates a tcp server by host name and port number. The first argument is the host name. The second argument is the port number. The third argument is the backlog.

6. Object Datagram

The *Datagram* class is a socket class used by udp socket. A datagram is constructed by the *UdpSocket accept* method. The purpose of a datagram is to store the peer information so one can reply to the sender. The datagram also stores in a buffer the data sent by the peer. This class does not have any constructor nor any specific method.

6.1. Predicate.

- datagram-p

6.2. Inheritance.

- Socket

7. Object UdpSocket

The *UdpSocket* class is a base class for all udp socket objects. The class is derived from the *Socket* class and provides some specific udp methods.

7.1. Predicate.

- udp-socket-p

7.2. Inheritance.

- Socket

7.3. Constructors.

- *UdpSocket* → *(none)*
  The *UdpSocket* constructor creates a new udp socket.

7.4. Methods.

- *accept* → *Datagram (none)*
  The *accept* method waits for an incoming datagram and returns a *Datagram* object. The datagram is initialized with the peer address and port as well as the incoming data.

8. Object UdpClient

The *UdpClient* class creates a udp client by host and port. The host argument can be either a name or an address object. The port argument is the server port to contact. The *UdpClient* class is derived from the *UdpSocket* class. This class has no specific methods.

8.1. Predicate.

- udp-client-p

8.2. Inheritance.

- UdpSocket

8.3. Constructors.

- *UdpClient → (String Integer)*
  The *UdpClient* constructor creates a new udp client by host and port. The first argument is the host name. The second argument is the port number.

9. Object UdpServer

The *UdpServer* class creates a udp server by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The *UdpServer* class is derived from the *UdpSocket* class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The host name can also be specified as the first argument, the port as second argument. Note that the host can be either a string or an address object.

9.1. Predicate.

- udp-server-p

9.2. Inheritance.

- UdpSocket

9.3. Constructors.

- *UdpServer → (none)*
  The *UdpServer* constructor creates a default udp server object.
- *UdpServer → (String—Address)*
  The *UdpServer* constructor creates a udp server object by host. The first argument is the host name or host address.
- *UdpServer → (String—Address Integer)*
  The *UdpServer* constructor creates a udp server object by host and port. The first argument is the host name or host address. The second argument is the port number.

10. Object Multicast

The *Multicast* class creates a udp multicast socket by port. An optional host argument can be either a name or an address object. The port argument is the server port to bind. The *Multicast* class is derived from the *UdpSocket* class. This class has no specific methods. With one argument, the server bind the port argument on the local host. The host name can also be specified as the first argument, the port as second argument. Note that the host can be either a string or an address object. This class is similar to the *UdpServer* class, except that the socket join the multicast group at construction and leave it at destruction.

10.1. Predicate.

- multicast-p

10.2. Inheritance.

- UdpSocket

10.3. Constructors.

- *Multicast → (String—Address)*
  The *Multicast* constructor creates a multicast socket object by host. The first argument is the host name or host address.
- *Multicast → (String—Address Integer)*
  The *Multicast* constructor creates a multicast socket object by host and port. The first argument is the host name or host address. The second argument is the port number.

CHAPTER 9

Standard Network Working Group Module

The *Standard Network Working Group* module is an original implemtation of the recommendations proposed by the NWG and currently found in the form of *Request for Comments* (RFC). Most of the objects are used with networking application, with the most common one beeing the *Universal Resource Identifier* (URI) object.

1. The uri class

The *Uri* class is a base class that parses a *Uniform Resource Identifier* or uri string and provides methods to access individual component of that uri. The implementation conforms to RFC 3986. The URI components are the scheme, the authority, the path, the query and the fragment. The class also takes care of the character escaping.

```
1 const uri (afnix:www:Uri "http://www.afnix.org")
```

An uri can be broken into several components called the *scheme* , the *authority* , the *path* , optionally the *query* and the *fragment* . The *Uri* class provide a method to retrieve each component of the parsed uri.

```
1 const uri (afnix:www:Uri "http://www.afnix.org/")
2 println (uri:get-scheme) # http
3 println (uri:get-authority) # www.afnix.org
4 println (uri:get-path) # /
```

1.1. Character conversion. The *Uri* class performs automatically the character conversion in the input uri. For example, the `+` character is replaced by a blank. The `%` character followed by two hexadecimal values is replaced by the corresponding ASCII character. Note that this conversion does now apply to the query string.

1.2. Query string. The *get-query* method returns the query string of the uri. The query string starts after the `?` character. The query string is a series of key-pair values separated by the `&` character.

```
1 const uri (afnix:www:Uri
2   "http://www.afnix.org?name=hello&value=world")
3 println (uri:get-query) # name=hello&value=world
```

The module also provides the *UriQuery* class that parses the query string and store the result in the form of a property list. The query string parse is particularly useful when writing automated scripts.

```
1 # create a query string object
2 const qs (afnix:nwg:UriQuery (uri:get-query))
3 # get the name value
4 qs:get-value "name"
```

2. Managing a cgi request

Managing a cgi request involves primarily the parsing of the requesting uri. The uri generally contains the http referrer as well as parameter which are stored in the form of a query string. However, depending on the cgi method which can be of type *GET* or *POST* , the treatment is somewhat different.

2.1. Checking the protocol version. In the presence of a cgi protocol, it is always a good idea to check the protocol version, or at least to put an assertion. The protocol version is normally *CGI/1.1* and is stored in the *GATEWAY_INTERFACE* environment variable.

```
# check the cgi protocol
assert "CGI/1.1" (
  afnix:sys:get-env "GATEWAY_INTERFACE")
```

2.2. Getting the query string. If the request method is `GET` , then the query string is available in the environment variable `QUERY_STRING` . If the request method is `POST` , the query string is available in the input stream. The length of the query string is given by the `CONTENT_LENGTH` environment variable. The following example illustrates the extraction of the query string.

```
# check the cgi protocol
assert "CGI/.1" (
  afnix:sys:get-env "GATEWAY_INTERFACE")
# initialize the query string
const query (afnix:sys:get-env "QUERY_STRING")
# get the request method
const rqm (afnix:sys:get-env "REQUEST_METHOD")
# check for a post request and update the query string
if (== rqm "POST") {
  # create a buffer from the content length
  const len (
    Integer (afnix:sys:get-env "CONTENT_LENGTH"))
  # get the standard input stream and read content
  const is (interp:get-input-stream)
  const buf (is:read len)
  # set the query string
  query:= (buf:to-string)
}
```

2.3. Parsing the query string. The *UriQuery* class is designed to parse a cgi query string. Once the string has been parsed, it is possible to perform a query by key since the class operates with a property list.

```
const query (
  afnix:www:UriQuery "name=hello&value=world")
query:length # 2
query:get-value "name" # hello
query:get-value "value" # world
```

The *UriQuery* class is the foundation to build cgi script. When the library is combined with the *web application management (wam)* service, powerful applications can be built easily.

3. Special functions

Several dedicated functions are available in the library as a way to ease the object manipulations. Theses functions operate mostly on uri and files as described below.

3.1. Uri functions. Several functions are designed to ease the uri manipulation. Most of them operate on the uri name or their associated system name. The *normalize-uri-name* function normalizes a string argument by adding a uri scheme if missing in the original string. If the function detects that the name starts with a host name, the *http* scheme is added. If the function detects that the string starts with a path, the *file* scheme is added. otherwise, the name argument is left untouched. The *system-uri-name* function normalizes the string argument by prioritizing the system name. The function attempts to find a file that match the sring argument and eventually build a uri file scheme. If the file is not fond, the normalization process occurs with the *normalize-uri-name* function.

```
# normalize a uri name
trans unm "http://www.afnix.org"
assert unm (
  afnix:nwg:normalize-uri-name unm)
assert unm (
  afnix:nwg:normalize-uri-name "www.afnix.org")
assert unm (
  afnix:nwg:normalize-uri-name "//www.afnix.org")
```

3.2. Mime functions. Mime functions are dedicated to easee the mainpulation of media types or mime. A media type is defined by a string in the form of a type and content value such as *text/plain* . The *mime-value-p* predicate returns true if a string mime value is a valid media type. From a file perspective, the *mime-extension-p* predicate returns true if the string extension has a valid media type associated to it. Finally, the *extension-to-mime* function can be used to get the string mime value associated with a file extension.

```
# check a media type
assert true (afnix:nwg:mime-value-p "text/plain")
# check the mime extension predicate
assert true (afnix:nwg:mime-extension-p "txt")
# check the extension to mime
assert "text/plain" (
  afnix:nwg:extension-to-mime "txt")
```

4. HTTP transaction objects

The concept of HTTP transactions is defined in RFC 2616. In the client/server approach, a client issues a request which is answered with a response. A special case arise when the server is asked to perform some extra works, such like executing a script. In this case, the answer is called a reply which is formatted into a response when the server does its job correctly.

The nature of the HTTP objects determines how the associated stream behaves. With a HTTP request, the object is filled by reading an input stream when operating on the server side. On the other hand, the request is filled by data when operating on the client side. With a HTTP response, the opposite situation occurs. The HTTP response is filled by reading an input stream when operating on the client side and filled by data when operating on the server side.

4.1. HTTP protocol. The *HttpProto* class is a base class designed to handle a HTTP header that is found in both HTTP request and response. The class is built around a property list that is filled either by parsing an input stream or by processing specific methods. The *HttpProto* defines also some methods which are often used with a HTTP request or response.

5. HTTP response

The *HttpResponse* class is a class designed to handle a HTTP response. When operating on the client side, the response object is built by reading an input stream. When operating on the server side, the response object is built by calling specific methods.

5.1. Creating a server response. A server response is created by specifying the response status code. By default, a HTTP response is created with the default media type *text/html* . If the media type needs to be changed, it can be passed as the second argument to the response constructor. By default, the empty constructor creates an empty constructor with a valid status code.

```
#create a valid response
const hr (afnix:nwg:HttpResponse 200)
```

Once the server response is created, it can be augmented with some headed values. Typically, a server will add some information about the response, such like the content length, the modification time or a tag. The *HttpResponse* provides several methods that ease the generation of these header values.

5.2. Creating a client response. A client response is created by binding an input stream to a response object. During the construction, the input stream is read and the HTTP protocol header is filled. It is also during this phase that the status code is processed. It is therefore important to ensure that a response object is built correctly before attempting to access it.

```
# create a client response by stream
const hr (afnix:nwg:HttpResponse is)
```

5.3. Reading a client response. When the response has been created, it is important to check its status code. Most of the time, the response is valid and its content can be read directly. The *status-ok-p* predicate returns true if the status code is valid. In such case, a HTTP stream can be built in order to read the response.

```
# check that a response is valid
if (hr:status-ok-p) {
  # create a http stream
  const rs (afnix:nwg:HttpStream ht is)
  # read the response stream
  while (rs:eos-p) (rs:read)
}
```

Before reading a http stream, it is important to detect and verify the nature of the response content. The *media-type-p* predicate returns true if the media type is defined and the *get-media-type* method returns the response type in the form of a mime code such like *text/html* . Eventually, the character set associated with the media type can also be detected. The *encoding-mode-p* predicate and the *get-encoding-mode* method can be used to detect the content encoding mode. However, it is worth to note that the *HttpStream* object is automatically sets with the proper encoding if it can be found in the response header.

5.4. Special client response. Certain response can sometime contains special status codes that require a specific treatment. This is the case when the response corresponds to a http redirection. In this case, the new uri must be fetched to get the desired response. The *location-p* predicate returns true if the response corresponds to a http redirect and the *get-location* method can be used to get the new location uri. If this situation arises, it is up to the implementation to decide what to do with the new uri. In most cases, a new request will be sent to the server.

6. Cookie object

The *Cookie* object is a special object that can be used during a http session, to post data to the http client. The idea behind *cookies* is to be able to maintain some state, during the user session for some time. A cookie is a *name/value* pair and eventually an expiration time. By default, the cookie object are defined for one http client session, but this behavior can be changed.

6.1. Managing cookies. A cookie is created with a *name/value* pair and eventually an expiration time. Such expiration time is called the *maximum-age* and is automatically formatted by the object. With two arguments a session cookie is created. With a third argument as an integer, the constructor set the maximum age in seconds.

```
1 # create a cookie with name/value
2 const cookie (afnix:nwg:Cookie "cartid" "123456789")
```

The cookie implementation follows the recommendation of the RFC-2965 for http state management. The most important point to remember is the interpretation of the maximum age that differs from one cookie version to another. With version 1, which is the default, the maximum age is defined relatively in seconds, while it is absolute with version 0.The maximum age is set either at construction or with the *set-max-age* method. The *set-max-age* method sets the cookie life time in seconds, in reference to the current time. A negative value is always reset to -1 and defined a session cookie. A 0 value tells the http client to remove the cookie. The *set-path* method defines the path for which this cookie apply.

6.2. Adding a cookie. Once the cookie is defined, the *set-cookie* method of the *HttpResponse* object can be used to install the cookie. Combined with the *write* method, the cookie can be send to the http client.

CHAPTER 10

Standard Network Working Group Reference

1. Object Uri

The *Uri* class is a base object used to parse or build a uniform resource identifier as defined by RFC 3986. The URI can be built by specifying each component or by parsing a string. When a string is given in the constructor, the class parses the string and extract all components. The uri components are the scheme, the authority, the path, the query and the fragment. The class also takes care of the character escaping.

1.1. Predicate.

- uri-p

1.2. Inheritance.

- Object

1.3. Constructors.

- *Uri → (none)*
  The *Uri* constructor creates an empty uri object.
- *Uri → (String)*
  The *Uri* constructor create a uri object by value. The string argument is the uri to parse at the object construction.
- *Uri → (String String Integer)*
  The *Uri* constructor create a uri object by scheme host and port. The first argument is the uri scheme. The second argument is the uri host name. The third argument is the uri port. The uri base name can be reconstructed from this information.

1.4. Methods.

- *parse → none (String)*
  The *parse* method reset the uri object, parse the string argument and fill the uri object with the result.
- *get-scheme → String (none)*
  The *get-scheme* method returns the scheme of the parsed uri object.
- *get-authority → String (none)*
  The *get-authority* method returns the authority part of the parsed uri.
- *get-path → String (none)*
  The *get-path* method returns the path of the parsed uri.
- *get-path-target → String (none)*
  The *get-path-target* method returns the path target of the parsed uri. The path target is the last element of the uri path.
- *get-query → String (none)*
  The *get-query* method returns the complete query string of the parsed uri. Note that characters are not escaped when getting the string.
- *get-fragment → String (none)*
  The *get-fragment* method returns the complete query string of the parsed uri.
- *get-base → String (none)*
  The *get-base* method returns the combined uri scheme and authority.
- *get-rname → String (none)*
  The *get-rname* method returns the reference uri name with the combined uri scheme, authority and path all percent encoded.
- *get-hname → String (none)*
  The *get-hname* method returns the combined uri scheme, authority and path.

- *get-aname → String (none)*
  The *get-aname* method returns the almost combined uri name with the scheme, authority, path and query.
- *add-path → Uri (String)*
  The *add-path* method adds a path to the calling uri and returns a new uri with the new path added to the old one.
- *get-href → Uri (String)*
  The *get-href* method returns a new uri by eventually combining the string argument. If the string argument correspond to an uri, the corresponding uri is built. Otherwise, the string argument is considered as a path to be added to the current uri in order to build a new uri.
- *get-system-path → String (none)*
  The *get-system-path* method returns the system path representation of the uri path. This function works only if the scheme if a file scheme.
- *get-path-encoded → String (none)*
  The *get-path-encoded* method returns the uri in the encoded form. Normally the *get-path* removes the percent-encoded characters which might not be appropriate with some protocol such like the http protocol. The *get-path-encoded* returns the original path. Note that getting the path with *getpath* and doing a percent coding might result in a different result since the internal representation uses normalized string.
- *get-host → String (none)*
  The *get-host* method returns the authority or path host name if any can be found with respect to the scheme. With a ftp, http or https scheme, the host is extracted from the authority. With a mailto scheme, the host is extracted from the path.
- *get-port → Integer (none)*
  The *get-port* method returns the authority port if any can be found with respect to the scheme.

2. Object UriQuery

The *UriQuery* class is a simple class that parses a uri query string and build property list. during the parsing process, a special transliteration process is done as specified by RFC 3986. This class is primarily used with *cgi* scripts. Note that the string to parse is exactly the one produced by the *get-query* method of the *Uri* class.

2.1. Predicate.

- uri-query-p

2.2. Inheritance.

- Plist

2.3. Constructors.

- *UriQuery → (none)*
  The *UriQuery* constructor creates an empty uri query object.
- *UriQuery → (String)*
  The *UriQuery* constructor create a uri object by value. The string argument is the uri query string to parse at the object construction. The query string is the one obtained from the *get-query* method of the *Uri* class.

2.4. Methods.

- *parse → none (String)*
  The *parse* method reset the uri query object, parses the string argument and fill the property list object with the result.
- *get-query → String (none)*
  The *get-query* method returns the original query string.

3. Object UriPath

The *UriPath* class is a class designed for the management of file system path associated with a uri. Typically, this class will be used with a http server or client when an association between a uri and a file name needs to be made. The general operation principle is to associate a path with a uri authority. The uri path is then concatanated to produce a new path. If the uri path is empty, it can be eventually replaced by a file name, known as the diretory index in the http terminology.

3.1. Predicate.

- uri-path-p

3.2. Inheritance.

- Object

3.3. Constructors.

- *UriPath → (none)*
  The *UriPath* constructor creates an empty uri path object.
- *UriPath → (String)*
  The *UriPath* constructor create a uri object by root path. The string argument is the uri root path.
- *UriPath → (String String)*
  The *UriPath* constructor create a uri object by root and index. The first string argument is the uri root path and the second string argument is the directory index path.
- *UriPath → (String String String)*
  The *UriPath* constructor create a uri object by root, index and authority. The first string argument is the uri root path, the second string argument is the directory index path and the third argument is the authority.

3.4. Methods.

- *get-root → String (none)*
  The *get-root* method returns the root path.
- *get-index → String (none)*
  The *get-index* method returns the index path.
- *get-authority → String (none)*
  The *get-authority* method returns the uri authority.
- *map-request-uri → String (none)*
  The *map-request-uri* map a request uri into a system path. The string argument is the request uri. The request uri must be an absolute path. The result string is the system path build with the root path.
- *normalize → String (none)*
  The *normalize* method build a system from a request path. The request path is associated with the root path and then normalized to produce a complete system path.

4. Object HttpProto

The *HttpProto* class is a base class that ease the deployment of the http protocol. The base class is built with a property list which is used to define the message header. The class also defines the write methods which are used to write a message either on an output stream or into a buffer.

4.1. Predicate.

- http-proto-p

4.2. Inheritance.

- Object

4.3. Methods.

- *reset → none (none)*
  The *reset* method resets the http protocol object by clearing the protocol version and header.
- *parse → none (none)*
  The *parse* method parse the input stream bound to the http protocol. In order to operate, an input stream must be associated with the protocol object or an exception is raised. After a stream has been parsed, the protocol version and the header are set.
- *write → none (none—OutputStream—Buffer)*
  The *write* method formats and writes the http protocol object to an output stream or a buffer. Without argument, the default output stream is used. With an argument, an output stream or a buffer object can be used.
- *header-length → Integer (none)*
  The *header-length* method returns the number of properties in the header.
- *header-exists-p → Boolean (String)*
  The *header-exists-p* predicate returns true if the property exists in the header. The string argument is the property name.
- *header-set → none (String Literal)*
  The *header-set* method sets a new property to the http header. The first argument is the property name. The second argument is a literal object which is internally converted to a string.
- *header-get → Property (Integer)*
  The *header-get* method returns a property object by index.
- *header-map → String (String)*
  The *header-map* method returns a property value by name. The string argument is the property name.
- *header-find → Property (String)*
  The *header-find* method returns a property object by name. The string argument is the property name. If the property is not found, the nil object is returned.
- *header-lookup → Property (String)*
  The *header-lookup* method returns a property object by name. The string argument is the property name. If the property is not found, an exception is raised.
- *header-plist → Plist (none)*
  The *header-plist* method returns the header in the form of a property list.
- *content-length-p → Boolean (none)*
  The *content-length-p* predicate returns true if the content length is defined in the protocol header.

- *get-content-length → Integer (none)*
  The *get-content-length* method returns the content length defined in the protocol header. If the content length is not defined in the header, the null value is returned.

- *media-type-p → Boolean (none)*
  The *media-type-p* predicate returns true if the content type is defined in the protocol header.
- *get-media-type → String (none)*
  The *get-media-type* method returns the media type defined in the protocol header. If the media type is not defined in the header, the default media type is returned.
- *encoding-mode-p → Boolean (none)*
  The *encoding-mode-p* predicate returns true if the encoding mode is defined in the protocol header.
- *get-encoding-mode → String (none)*
  The *get-encoding-mode* method returns the protocol encoding mode. If the encoding mode is not defined in the protocol header, the default encoding mode is returned.

5. Object HttpRequest

The *HttpRequest* class is a base class designed to handle a http request. The class operates with the protocol version 1.1 as defined by RFC 2616. For a server request, the request is built by reading an input stream and setting the request command with its associated header. For a client request, the request is formatted with a request command and a eventually a uri. In both cases, the header is filled automatically depending on the request side.

5.1. Predicate.

- http-request-p

5.2. Inheritance.

- HttpProto

5.3. Constructors.

- *HttpRequest → (none)*
  The *HttpRequest* constructor creates a default http request. By default, the request object is built with the *GET* method and the request uri set to the root value.
- *HttpRequest → (String)*
  The *HttpRequest* constructor creates a http request object with a specific command. By default, the request uri is set to root, except for the *OPTIONS* method
- *HttpRequest → (Uri)*
  The *HttpRequest* constructor creates a http request object with a uri. The default request method is *GET* .
- *HttpRequest → (InputStream)*
  The *HttpRequest* constructor creates a http request object with a specific input stream. At construction, the request header is cleared and the input stream is bound to the object.
- *HttpRequest → (String String)*
  The *HttpRequest* constructor creates a http request object with a specific method and a uri name. The first string argument is the request method to use. The second string argument is the uri attached to the command. Note that the term *uri* should be understood as a *request uri* .
- *HttpRequest → (String Uri)*
  The *HttpRequest* constructor creates a http request object with a specific method and a uri. The first string argument is the request method to use. The second argument is the uri attached to the method.

5.4. Methods.

- *set-method → none (String)*
  The *set-method* method sets the request method. This method does not check that the command is a valid HTTP method and thus leaves plenty of room for server development. As a matter of fact, RFC 2616 does not prohibit the existence of such extension.
- *get-method → String (none)*
  The *get-method* method returns the request method string.
- *set-uri → none (String)*
  The *set-uri* method sets the request uri. The argument string does not have to be a valid uri string since some commands might accept special string such like "*" to indicate all applicable uri.

- *get-uri* → *String (none)*
  The *get-uri* method returns the request uri string.

6. Object HttpResponse

The *HttpResponse* class is a base class designed to handle a http response. The class operates with the protocol version 1.1 as defined by RFC 2616. For a client response, the response is built by reading an input stream and setting the response status code with its associated header. For a server response, the response is formatted with a response status and additional header information. In both cases, the header is filled automatically depending on the response side. On the other hand, trying to set some header with an input stream bound to the response object might render the response object unusable.

6.1. Predicate.

- http-response-p

6.2. Inheritance.

- HttpProto

6.3. Constructors.

- *HttpResponse → (none)*
  The *HttpResponse* constructor creates a default http response object. The response is marked valid with a default *text/plain* media type.
- *HttpResponse → (Integer)*
  The *HttpResponse* constructor creates a http response object with a status code. The response code is associated with the default *text/plain* media type.
- *HttpResponse → (InputStream)*
  The *HttpResponse* constructor creates a http response object with a specific input stream. At construction, the response header is cleared and the input stream is bound to the object.
- *HttpResponse → (Integer String)*
  The *HttpResponse* constructor creates a http response object with a status code and a media type. The first argument is the status code. The second argument is the associated media type.

6.4. Methods.

- *set-status-code → none (Integer)*
  The *set-status-code* method sets the response status code.
- *get-status-code → Integer (none)*
  The *get-status-code* method returns the response status code.
- *map-status-code → String (none)*
  The *map-status-code* method returns a string representation of the response status code.
- *status-ok-p → Boolean (none)*
  The *status-ok-p* predicate returns true if the response status code is valid (aka status 200).
- *status-error-p → Boolean (none)*
  The *status-error-p* predicate returns true if the response status code is an error code.
- *location-p → Boolean (none)*
  The *location-p* predicate returns true is the response status code indicates that a request should be made at another location. The location can be found with the *get-location* method.
- *get-location → String (none)*
  The *get-location* method returns the location uri found in the response header. This method is equivalent to a header query.

- *set-location → none (String)*
  The *set-location* method set the redirect location in the response header. The string argument is the location uri.
- *set-cookie → none (Cookie)*
  The *set-cookie* method sets a cookie object to the http header. The cookie version is properly handled by the method.

7. Object Cookie

The *Cookie* class is a special class designed to handle cookie setting within a http transaction. A cookie is *name/value* pair that is set by the server and stored by the http client. Further connection with the client will result with the cookie value transmitted by the client to the server. A cookie has various parameters that controls its existence and behavior. The most important one is the *cookie maximum age* that is defined in seconds. A null value tells the client to discard the cookie. A cookie without maximum age is valid only during the http client session. A cookie can be added to the *HttpReply* object with the *set-cookie* method. A cookie can be constructed with a *name/value* pair. An optional third argument is the maximum age. The default cookie version is 1 as specified by RFC 2965. With a version 1, the maximum age is interpreted as the number of seconds before the cookie expires. With version 0, the maximum age is the absolute time.

7.1. Predicate.

- cookie-p

7.2. Inheritance.

- Object

7.3. Constructors.

- *Cookie → (String String)*
  The *Cookie* constructor creates a cookie with a name value pair. The first argument is the cookie name. The second argument is the cookie value.
- *Cookie → (String String Integer)*
  The *Cookie* constructor creates a cookie with a name value pair and a maximum age. The first argument is the cookie name. The second argument is the cookie value. The third argument is the cookie maximum age.

7.4. Methods.

- *get-version → Integer (none)*
  The *get-version* method returns the cookie version.
- *set-version → none (Integer)*
  The *set-version* method sets the cookie version. The version number can only be 0 or 1.
- *get-name → String (none)*
  The *get-name* method returns the cookie name. This is the name store on the http client.
- *set-name → none (String)*
  The *set-name* method sets the cookie name. This is the name store on the http client.
- *get-value → String (none)*
  The *get-value* method returns the cookie value. This is the value stored on the http client bounded by the cookie name.
- *set-value → none (String)*
  The *set-value* method sets the cookie value. This is the value store on the http client bounded by the cookie name.
- *get-maximum-age → Integer (none)*
  The *get-maximum-age* method returns the cookie maximum age. The default value is -1, that is, no maximum age is set and the cookie is valid only for the http client session.

- *set-maximum-age → none (Integer)*
  The *set-maximum-age* method sets the cookie maximum age. A negative value is reset to -1. A 0 value tells the http client to discard the cookie. A positive value tells the http client to store the cookie for the remaining seconds.
- *get-path → String (none)*
  The *get-path* method returns the cookie path value. The path determines for which http request the cookie is valid.
- *set-path → none (String)*
  The *set-path* method sets the cookie path value. The path determines for which http request the cookie is valid.
- *get-domain → String (none)*
  The *get-domain* method returns the cookie domain value.
- *set-domain → none (String)*
  The *set-domain* method sets the cookie domain value. It is string recommended to use the originator domain name since many http client can reject cookie those domain name does not match the originator name.
- *get-port → Integer (none)*
  The *get-port* method returns the cookie port number.
- *set-port → none (Integer)*
  The *set-port* method sets the cookie port number. This value is not used with a cookie version 0.
- *get-comment → String (none)*
  The *get-comment* method returns the cookie comment value.
- *set-comment → none (String)*
  The *set-comment* method sets the cookie comment value.
- *get-comment-url → String (none)*
  The *get-comment-url* method returns the cookie comment url value.
- *set-comment-url → none (String)*
  The *set-comment-url* method sets the cookie comment url value. This value is not used with cookie version 0.
- *get-discard → Boolean (none)*
  The *get-discard* method returns the cookie discard flag.
- *set-discard → none (Boolean)*
  The *set-discard* method sets the cookie discard flag. The discard flag the tells the user agent to destroy the cookie when it terminates. This value is not used with cookie version 0.
- *get-secure → Boolean (none)*
  The *get-secure* method returns the cookie secure flag.
- *set-secure → none (Boolean)*
  The *set-secure* method sets the cookie secure flag. When a cookie is secured, it is only returned by the http client if a connection has been secured (i.e use https).
- *to-string → String (none)*
  The *to-string* method returns a string formatted for the http reply header. Normally this method should not be called since the *set-cookie* method of the *httpReply* takes care of such thing.

8. Object Session

The *Session* class is a class that defines a session to be associated with a transaction. The session object is designed to be persistent so that its data information can be retreived at any time. A session object has also the particularity to have a limited lifetime. A session object is created by name with an identifier. The session object is designed to hold a variety of parameters that are suitable for both the authentication and the session lifetime. A session is primarily defined by name with an optional information string. The session is generally associated an authentication visa which contains the session identity. The visa provides a secure mechanism compatible with a single sign on session. A session key is automatically generated when the session is created. Such key is used to generate a session hash id which can be used as a cookie value. The cookie name is also stored in the session object. When a cookie is generated, the session hash name is combined with the session hash id for the cookie production.

8.1. Predicate.

- session-p

8.2. Inheritance.

- Taggable

8.3. Constructors.

- *Session → (String)*
  The *Session* constructor creates a session by name. The string argument is the session name.
- *Session → (String String)*
  The *Session* constructor creates a session with a name and a user. The first argument is the session name. The second argument is the session information..
- *Session → (String String Integer)*
  The *Session* constructor creates a session with a name, a user and a maximum age. The first argument is the session name. The second argument is the session infor-martion. The third argument is the session maximum age expressed in seconds.

8.4. Methods.

- *expire-p → Boolean (none)*
  The *expire-p* predicate returns true if the session has expired.
- *set-hash-id → none (String)*
  The *set-hash-id* method sets the session hash identifier. The session hash id must be unique and secured enough so that the session name cannot be derived from it.

- *get-hash-id → String (none)*
  The *get-hash-id* method returns the session hash identifier.
- *set-path → none (String)*
  The *set-path* method sets the session path.
- *get-path → String (none)*
  The *get-path* method returns the session path.
- *get-max-age → Integer (none)*
  The *get-max-age* method returns the session maximum age.
- *set-max-age → none (Integer)*
  The *set-max-age* method sets the session maximum age. The maximum age is an integer in seconds relative to the current time. If the maximum age is set to 0, the session is closed.

- *get-remaining-time → Integer (none)*
  The *get-remaining-time* method returns the remaining valid session time.
- *get-expire-time → Integer (none)*
  The *get-expire-time* method returns the session expiration time in seconds. The expiration time is an absolute time.
- *set-expire-time → none (Integer)*
  The *set-expire-time* method sets the session expiration time. The expiration time is an absolute time in seconds.
- *get-creation-time → Integer (none)*
  The *get-creation-time* method returns the session creation time. The creation time is an absolute time in seconds.
- *get-modification-time → Integer (none)*
  The *get-modification-time* method returns the session creation time. The modification time is an absolute time in seconds.
- *get-cookie → Cookie (name)*
  The *get-cookie* method bakes a session cookie. The string argument is the cookie name those value is the session hash id value.
- *close → Cookie (name)*
  The *close* method close a session by reseting the session maximum age to 0. The method returns a cookie that can be used for closing the session on the peer side. The string argument is the cookie name those value is the session hash id value.

8.5. Functions.

- *mime-extension-p → Boolean (String)*
  The *mime-extension-p* predicates returns true if a media type extension - mime extension - is defined. Most of the time, media type extension can be seen as a file extension.
- *mime-value-p → Boolean (String)*
  The *mime-value-p* predicates returns true if a media type - mime value - is defined.

- *extension-to-mime → String (String [Boolean])*
  The *extension-to-mime* function converts a media type extension into a media type. In the first form, without a second argument, if the media type extension does not exist, an exception is raised. In the second form, with the second argument set to true, if the media type extension does not exist, the default media type is returned. If the flag is set to false, an exception is raised like the first form.
- *string-uri-p → Boolean (String)*
  The *string-uri-p* predicates returns true if the string argument is a uri.
- *normalize-uri-name → String (String)*
  The *normalize-uri-name* function normalizes the string argument by adding a uri scheme if missing in the original string. If the function detects that the name starts with a host name, the "http" scheme is added. If the function detects that the string starts with a path, the "file" scheme is added. otherwise, the name argument is left untouched.
- *system-uri-name → String (String)*
  The *system-uri-name* function normalizes the string argument by prioritizing the system name. The function attempts to find a file that match the string argument and eventually build a uri file scheme. If the file is not fond, the normalization process occurs with the *normalize-uri-name* function.
- *path-uri-name → String (String)*
  The *path-uri-name* function normalizes the string argument by extracting a path

associated with the uri string. If the string is a valid uri string, the path is the uri path component. If the uri path is empty, it is normalized to a /. If the string argument is not a uri string, the string is assumed to be a partial uri and both query and fragment parts are removed if present.

- *normalize-uri-host → String (String)*
  The *normalize-uri-host* function normalizes the string argument uri host name. This function is useful with certain class of host representation which uses extra characters.
- *normalize-uri-port → String (String)*
  The *normalize-uri-port* function normalizes the string argument uri port value. This function is useful with certain class of host representation which uses extra characters.

CHAPTER 11

Standard Security Module

The *Standard Security* module is an original implementation of several standards and techniques used in the field of cryptography. The module provides the objects than enables message hashing, symetric and assymetric ciphers and digital signature computation. The implementation follows the recommendation from NIST and PKCS and the standard reference that it implements is always attached to the underlying object.

1. Hash objects

Hashing is the ability to generate an *almost* unique representation from a string. Although, there is no guarantee that two different string will not produce the same result – known as a collision – the sophistication of the hashing function attempt to minimize such eventuality. The hashing process is not reversible. There are several hashing functions available in the public domain. To name a few, MD5 is the *message digest 5* , and SHA is the *secure hash algorithm* . The following table illustrates the size of the result with different hashing functions.

1.1. Hasher object. The *Hasher* class is a text hashing computation class. The class computes a *hash value* from a literal object, a buffer or an input stream. Once computed, the hash value is stored as an array of bytes that can be retrieved one by one or at all in the form of a string representation.

1.2. Creating a hasher. Several hasher objects are available in the module. For example, the *Md5* object is the hasher object that implements the MD-5 algorithm. The constructor does not take any argument.

```
# get a MD-5 hasher
const md (afnix:sec:Md5)
# check the object
afnix:sec:hasher-p md # true
```

The *compute* method computes the hash value. For example, the string "abc" returns the value `"900150983CD24FB0D6963F7D28E17F72"` which is 16 bytes long.

```
const hval (md:compute "abc")
```

Function	Result size
MD-2	128 bits
MD-4	128 bits
MD-5	128 bits
SHA-1	160 bits
SHA-224	224 bits
SHA-256	256 bits
SHA-384	384 bits
SHA-512	512 bits

Hasher	Size	Constructor
SHA-1	160 bits	Sha1
SHA-224	224 bits	Sha224
SHA-256	256 bits	Sha256
SHA-384	384 bits	Sha384
SHA-512	512 bits	Sha512

Key	Description
KSYM	Symmetric cipher key
KRSA	Asymmetric RSA cipher key
KMAC	Message authentication key
KDSA	Message signature key

Key	Type	Description
KSYM	byte	Byte array size
KRSA	bits	Modulus size
KMAC	byte	Byte array size
KDSA	bits	Signature size

1.3. Creating a SHA hasher. There are several SHA objects that produces results of different size as indicated in the next table.

The *compute* method computes the hash value. For example, the string "abc" returns with SHA-1 the 20 bytes long value:

```
"A9993E364706816ABA3E25717850C26C9CD0D89D"
```

2. Cipher key principles

Cipher key management is an important concept in the ciphering land. In a simple mode, a key is used by a cipher to encode some data. Although the key can be any sequence of bytes, it is preferable to have the key built from a specific source such like a pass-phrase. A cipher key comes basically into two flavors: keys for symmetric ciphers and keys for asymmetric ciphers. A key for a symmetric cipher is easy to derive and generally follows a standard process which is independent of the cipher itself. A key for an asymmetric cipher is more complex to derive and is generally dependent on the cipher itself.

2.1. Key operations. The basic operations associated with a key are the key identification by type and size. The key type is an item that identifies the key nature. The *get-type* method returns the key type as specified by the table below.

The message authentication key as represented by the *KMAC* symbol is similar to the symmetric key. The key type can be obtained with the *get-type* method.

```
# get the key type
const type (key:get-type)
```

The key size is the canonical size as specified by the key or the cipher specification. The *get-bits* returns the key size in bits. The *get-size* returns the key size in bytes rounded to the nearest value. The table below describes the nature of the key size returned.

```
const bits (key:get-bits)
const size (key:get-size)
```

Key	Argument	Description
KSYM	none	Symmetric key octet string
KRSA	RSA-MODULUS	RSA modulus octet string
KRSA	RSA-PUBLIC-EXPONENT	RSA public exponent octet string
KRSA	RSA-SECRET-EXPONENT	RSA secret exponent octet string
KMAC	none	Message authentication key octet string
KDSA	DSA-P-PRIME	DSA secret prime octet string
KDSA	DSA-Q-PRIME	DSA secret prime octet string
KDSA	DSA-SECRET-KEY	DSA secret key
KDSA	DSA-PUBLIC-KEY	DSA public key
KDSA	DSA-PUBLIC-GENERATOR	DSA public generator

2.2. Key representation. Unfortunately, it is not easy to represent a key, since the representation depends on the key's type. For example, a symmetric key can be formatted as a simple octet string. On the other hand, a RSA key has two components; namely the modulus and the exponent, which needs to be distinguished and therefore making the representation more difficult. Other cipher keys are even more complicated. For this reason, the representation model is a relaxed one. The *format* method can be called without argument to obtain an unique octet string representation if this representation is possible. If the key representation requires some parameters, the format method may accept one or several arguments to distinguish the key components.

```
# get a simple key representation
println (key:format)
# get a rsa modulus key representation
println (key:format afnix:sec:Key:RSA-MODULUS)
```

There are other key representations. The natural one is the byte representation for a symmetric key, while a number based representation is generally more convenient with asymmetric keys. The *get-byte* method returns a key byte by index if possible. The *get-relatif-key* returns a key value by relatif number if possible.

3. Symmetric cipher key

3.1. Creating a symmetric cipher key. The *Key* class can be used to create a cipher key suitable for a symmetric cipher. By default a 128 bits random key is generated, but the key can be also generated from an octet string.

```
const key (afnix:sec:Key)
assert true (afnix:sec:key-p key)
```

The constructor also supports the use of an octet string representation of the key.

```
# create an octet string key
const key (afnix:sec:Key "0123456789ABCDEF")
assert true (afnix:sec:key-p key)
```

3.2. Symmetric key functions. The basic operation associated with a symmetric key is the byte extraction. The *get-size* method can be used to determine the byte key size. Once the key size has been obtained, the key byte can be accessed by index with the *get-byte* method.

```
# create a 256 random symmetric key
const key (afnix:sec:Key afnix:sec:Key:KSYM 256)
# get the key size
const size (key:get-size)
# get the first byte
const byte (key:get-byte 0)
```

4. Asymmetric cipher key

An asymmetric cipher key can be generated for a particular asymmetric cipher, such like RSA. Generally, the key contains several components identified as the public and secret key components. These components are highly dependent on the cipher type. Under some circumstances, all components might not be available.

4.1. Creating an asymmetric cipher key. The *Key* class can be used to create a specific asymmetric cipher key. Generally, the key is created by type and and bits size.

```
# create a 1024 bits rsa key
const key (afnix:sec:Key afnix:sec:Key:KRSA 1024)
```

An asymmetric cipher key constructor is extremely dependent on the cipher type. For this reason, there is no constructor that can operate with a pass-phrase.

4.2. Asymmetric key functions. The basic operation associated with a asymmetric key is the relatif based representation which is generally available for all key components. For example, in the case of the RSA cipher, the modulus, the public and secret exponents can be obtained in a relatif number based representation with the help of the *get-relatif-key* method.

```
# create a 512 rsa key
const key (afnix:sec:Key afnix:sec:Key:KRSA 512)
# get the key modulus
const kmod (
  key:get-relatif-key afnix:sec:Key:RSA-MODULUS)
# get the public exponent
const pexp (
  key:get-relatif-key afnix:sec:Key:RSA-PUBLIC-EXPONENT)
# get the secret exponent
const sexp (
  key:get-relatif-key afnix:sec:Key:RSA-SECRET-EXPONENT)
```

5. Message authentication key

5.1. Creating a message authentication key. The *Key* class can also be used to create a message authentication key suitable for a message authentication code generator or validator. By default a 128 bits random key is generated, but the key can be also generated from an octet string.

```
const key (afnix:sec:Key afnix:sec:Key:KMAC)
assert true (afnix:sec:key-p key)
```

The constructor also supports the use of an octet string as a key representation.

```
# create an octet string key
const key (
  afnix:sec:Key afnix:sec:Key:KMAC "0123456789ABCDEF")
assert true (afnix:sec:key-p key)
```

5.2. Message authentication key functions. The basic operation associated with a message authentication key is the byte extraction. The *get-size* method can be used to determine the byte key size. Once the key size has been obtained, the key byte can be accessed by index with the *get-byte* method.

```
# create a 256 random message authentication key
const key (afnix:sec:Key afnix:sec:Key:KMAC 256)
# get the key size
const size (key:get-size)
# get the first byte
const byte (key:get-byte 0)
```

5.3. Signature key functions. The basic operation associated with a signature key is the relatif based representation which is generally available for all key components. For example, in the case of the DSA signer, the prime numbers, the public and secret components can be obtained in a relatif number based representation with the help of the *get-relatif-key* method.

```
# create a 1024 dsa key
const key (afnix:sec:Key afnix:sec:Key:KDSA)
# get the key size
const size (key:get-size)
# get the secret component
const sexp (
  key:get-relatif-key afnix:sec:Key:DSA-SECRET-KEY)
```

6. Stream cipher

A stream cipher is an object that encodes an input stream into an output stream. The data are read from the input stream, encoded and transmitted onto the output stream. There are basically two types of stream ciphers known as symmetric cipher and asymmetric cipher.

6.1. Symmetric cipher. A symmetric cipher is a cipher that encodes and decode data with the same key. Normally, the key is kept secret, and the data are encoded by block. For this reason, symmetric cipher are also called block cipher. In normal mode, a symmetric cipher is created with key and the data are encoded from an input stream as long as they are available. The block size depends on the nature of the cipher. As of today, the recommended symmetric cipher is the *Advanced Encryption Standard* or AES, also known as Rijndael.

6.2. Asymmetric cipher. An asymmetric cipher is a cipher that encodes and decode data with two keys. Normally, the data are encoded with a public key and decoded with a private key. In this model, anybody can encode a data stream, but only one person can read them. Obviously, the model can be reverse to operate in a kind of signature mode, where only one person can encode the data stream and anybody can read them. Asymmetric cipher are particularly useful when operating on unsecured channels. In this model, one end can send its public key as a mean for other people to crypt data that can only be read by the sender who is supposed to have the private key. As of today, the recommended asymmetric ciphers are RSA and DH.

6.3. Serial cipher. A serial cipher is a cipher that encodes and decode data on a byte basis. Normally, the data are encoded and decoded with the same key, thus making the symmetric cipher key, the ideal candidate for a serial cipher key. Since the data is encoded on a byte basis, it can be used efficiently with a stream. However, the serial cipher does not define a block size and therefore require some mechanism to prevent a buffer overrun when reading bytes from a stream. For this reason, the serial cipher defines a default *serial*

block size that can be used to buffer the stream data. A method is provided in the class to control the buffer size and is by default set to 4Kib bytes.

6.4. Cipher base class. The *Cipher* base class is an abstract class that supports the symmetric, asymmetric and serial cipher models. A cipher object has a name and is bound to a key that is used by the cipher. The class provides some base methods that can be used to retrieve some information about the cipher. The *get-name* method returns the cipher name. The *set-key* and *get-key* methods are both used to set or retrieve the cipher key.

The cipher operating mode can be found with the *get-reverse* method. If the *get-reverse* method returns true, the cipher is operating in decoding mode. Note that a *set-reverse* method also exists.

7. Block cipher

A block cipher is an object that encodes an input stream with a symmetric cipher bound to a unique key. Since a block cipher is symmetric, the data can be coded and later decoded to their original form. The difference with the *Cipher* base class is that the *BlockCipher* class provides a *get-block-size* method which returns the cipher block size.

7.1. Block Cipher base. The *BlockCipher* class is a base class for the block cipher engine. The class implements the *stream* method that reads from an input stream and write into an output stream. The *BlockCipher* class is an abstract class and cannot be instantiated by itself. The object is actually created by using a cipher algorithm class such like the *Aes* class.

```
1 trans count (cipher:stream os is)
```

The *stream* method returns the number of characters that have been encoded. Care should be taken that most of the stream cipher operates by block and therefore, will block until a complete block has been read from the input stream, unless the end of stream is read. The block cipher is always associated with a padding scheme. By default, the NIST 800-38A recommendation is associated with the block cipher, but can be changed with the *set-padding-mode* .

7.2. Creating a block cipher. A *BlockCipher* object can be created with a cipher constructor. As of today, the *Advanced Encryption Standard* or AES is the recommended symmetric cipher. The *Aes* class creates a new block cipher that conforms to the AES standard.

```
1 const cipher (afnix:sec:Aes)
```

A block cipher can be created with a key and eventually a reverse flag. With one argument, the block cipher key is associated with the cipher. Such key can be created as indicated in the previous section. The reverse flag is used to determine if the cipher operate in encoding or decoding mode. By default, the cipher operates in coding mode.

```
1 # create a 256 bits random key
2 const key (afnix:sec:Key afnix:sec:KSYM 256)
3 # create an aes block cipher
4 const aes (afnix:sec:Aes key)
```

7.3. Block cipher information. The *BlockCipher* class is derived from the *Cipher* class and contains several methods that provide information about the cipher. This include the cipher block size with the *get-block-size* method.

```
1 println (aes:get-block-size)
```

8. Input cipher

In the presence of a *Cipher* object, it is difficult to read an input stream and encode the character of a block basis. Furthermore, the existence of various method for block padding makes the coding operation even more difficult. For this reason, the *InputCipher* class provides the necessary method to code or decode an input stream in various mode of operations.

8.1. Input cipher mode. The *InputCipher* class is an input stream that binds an input stream with a cipher. The class acts like an input stream, read the character from the bounded input stream and encode or decode them from the bended cipher. The *InputCipher* defines several modes of operations. In *electronic codebook mode* or ECB, the character are encoded in a block basis. In *cipher block chaining* mode, the block are encoded by doing an XOR operation with the previous block. Other modes are also available such like *cipher feedback mode* and *output feedback mode* .

8.2. Creating an input cipher. By default an input cipher is created with a cipher object. Eventually, an input stream and/or the input mode can be specified at the object construction.

```
# create a key
const key (afnix:sec:Key "hello world")
# create a direct cipher
const aes (afnix:sec:Aes key)
# create an input cipher
const ic (afnix:sec:InputCipher aes)
```

In this example, the input cipher is created in ECB mode. The input stream is later associated with the *set-is* method.

8.3. Input cipher operation. The *InputCipher* class operates with one or several input streams. The *set-is* method sets the input stream. Read operation can be made with the help of the *valid-p* predicate.

```
while (ic:valid-p) (os:write (ic:read))
```

Since the *InputCipher* operates like an input stream, the stream can be read as long as the *valid-p* predicate returns true. Note that the *InputCipher* manages automatically the padding operations with the mode associated with the block cipher.

9. Asymmetric cipher

A public cipher is an object that encodes an input stream with a asymmetric cipher bound to a public and secret key. In theory, there is no difference between a block cipher and a public cipher. Furthermore, the interface provided by the engine is the same for both objects.

9.1. Public cipher. A public cipher is an asymmetric stream cipher which operates with an asymmetric key. The main difference between a block cipher and a public cipher is the key nature as well as the encoded block size. With an asymmetric cipher, the size of the message to encode is generally not the same as the encoded block, because a message padding operation must occurs for each message block.

```
trans count (cipher:stream os is)
```

The *stream* method returns the number of characters that have been encoded. Like the block cipher, the *stream* method encodes an input stream or a buffer object. The number of encoded bytes is returned by the method.

Cipher	Padding mode	Default
RSA	PKCS 1.5, PKCS 2.1, ISO/IEC 18033-2	PKCS 1.5

Standard	Name
DSS	Digital Signature Standard
RSA	RSA based signature

9.2. Creating a public cipher. A *PublicCipher* object can be created with a cipher constructor. The *RSA* asymmetric cipher is the typical example of public cipher. It is created by binding a RSA key to it. For security reasons, the key size must be large enough, typically with a size of at lease 1024 bits.

```
const key (afnix:sec:Key afnix:sec:Key:KRSA 1024)
const rsa (afnix:sec:Rsa key)
```

A block cipher can be created with a key and eventually a reverse flag. Additional constructors are available to support various padding mode. Such padding mode depends on the cipher type. For example, the RSA cipher supports the ISO 18033-2 padding mode with a KDF1 or KDF2 object. Such constructor requires a hasher object as well.

```
# create a 1024 bits rsa key
const key (afnix:sec:Key afnix:sec:KRSA 1024)
# create a SHA-1 hasher
const ash (afnix:sec:Sha1)
# create a rsa public cipher
const rsa (afnix:sec:Rsa key ash "Demo")
# set the padding mode
rsa:set-padding-mode afnix:sec:Rsa:PAD-OAEP-K1
```

9.3. Public cipher padding mode. Like any cipher, a padding mode can be associated with the cipher. The *set-padding-mode* method can be used to set or change the padding mode. Depending on the padding mode type, additional objects might be needed at construction.

The default padding mode depends on the cipher type. For RSA, the default padding mode is set to PKCS 1.5 for compatibility reason.

10. Signature objects

A digital signature is a unique representation, supposedly non forgeable, designed to authenticate a document, in whatever form it is represented. For example, a signature is used to sign a certificate which is used during the process of establish a secured connection over the Internet. A signature can also be used to sign a *courrier* or keys as it is in the Openssh protocol. Digital signatures come into several flavors eventually associated with the signed document. Sometimes, the signature acts as a container and permits to retrieve the document itself. Whatever the method, the principle remains the same. As of today technology, there are two standards used to sign document as indicated below.

10.1. Signer and signature objects. The process of generating a signature is done with the help of a *Signer* object. A signer object is a generic object, similar in functionality to the hasher object. The result produced by a signer object is a *Signature* object which holds the generated signature.

Standard	Key	Signer
DSS	KDSA	Dsa

10.2. Signature key. The process of generating a signature often requires the use of a key. Such key can be generated with the help of the *Key* object. The nature of the key will depend on the target signature. The following table is a resume of the supported keys.

In the case of DSS, a key can be generated automatically, although this process is time consuming. The default key size is 1024 bits.

```
const key (afnix:sec:Key afnix:sec:Key:KDSA)
assert 1024 (key:get-bits)
```

10.3. Creating a signer. A *Signer* object is created with a particular signature object such like DSA. The *Dsa* object is a signer object that implements the *Digital Signature Algorithm* as specified by the *Digital Signature Standard (DSS)* in *FIPS-PUB 186-3* .

```
# create a dsa signer
const dsa (afnix:sec:Dsa key)
assert true (afnix:sec:dsa-p dsa)
```

10.4. Creating a signature. A signature is created with the help of the *compute* method. The *Signature* object is similar to the *Hasher* and operates with string or streams.

```
# create a signature object
const sgn (dsa:compute "afnix")
assert true (afnix:sec:signature-p sgn)
```

Once the signature is created, each data can be accessed directly with the associated component mapper. In the case of DSS, there are two components as show below.

```
# get the DSS S component
sgn:get-relatif-component
afnix:sec:Signature:DSA-S-COMPONENT
# get the DSS R component
sgn:get-relatif-component
afnix:sec:Signature:DSA-R-COMPONENT
```

CHAPTER 12

Standard Security Reference

1. Object Hasher

The *Hasher* class is a base class that is used to build a message hash. The hash result is stored in an array of bytes and can be retrieved byte by byte or as a formatted printable string. This class does not have a constructor.

1.1. Predicate.

- hasher-p

1.2. Inheritance.

- Nameable

1.3. Methods.

- *reset → none (none)*
  The *reset* method reset the hasher object with its associated internal states.
- *hash-p → Boolean (String)*
  The *hash-p* predicate returns true if the string argument is potentially a hash value. It is not possible, with our current technology, to reverse a hash value to one or several representations, nor it is possible to assert that such value exists.
- *get-byte → Byte (Integer)*
  The *get-byte* method returns the hash byte value by index. The argument is the byte index which must be in the range of the hash result length.
- *format → String (none)*
  The *format* method return a string representation of the hash value.
- *compute → String (Literal—Buffer—InputStream)*
  The *compute* method computes the hash value from a string, a buffer or an input stream. The method returns a string representation of the result hash value. When the argument is a buffer object or an input stream, the characters are consumed from the object.
- *derive → String (String)*
  The *derive* method computes the hash value from an octet string which is converted before the hash computation. The method returns a string representation of the result hash value.
- *get-hash-length → Integer (none)*
  The *get-hash-length* method returns the hasher length in bytes.
- *get-result-length → Integer (none)*
  The *get-result-length* method returns the hasher result length in bytes. The result length is less or equal to the hasher length and is set at construction.

2. Object Md2

The *Md2* class is a hashing class that implements the MD-2 algorithm.

2.1. Predicate.

- md2-p

2.2. Inheritance.

- Hasher

2.3. Constructors.

- *Md2* → *(none)*
  The *Md2* constructor creates a default hashing object that implements the MD-2 algorithm.
- *Md2* → *(Integer)*
  The *Md2* constructor creates a MD-2 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

3. Object Md4

The *Md4* class is a hashing class that implements the MD-4 algorithm.

3.1. Predicate.

- md4-p

3.2. Inheritance.

- Hasher

3.3. Constructors.

- *Md4* → *(none)*
  The *Md4* constructor creates a default hashing object that implements the MD-4 algorithm.
- *Md4* → *(Integer)*
  The *Md4* constructor creates a MD-4 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

4. Object Md5

The *Md5* class is a hashing class that implements the MD-5 algorithm.

4.1. Predicate.

- md5-p

4.2. Inheritance.

- Hasher

4.3. Constructors.

- *Md5* → *(none)*
  The *Md5* constructor creates a default hashing object that implements the MD-5 algorithm.
- *Md5* → *(Integer)*
  The *Md5* constructor creates a MD-5 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

5. Object Sha1

The *Sha1* class is a hashing class that implements the SHA-1 algorithm.

5.1. Predicate.

- sha1-p

5.2. Inheritance.

- Hasher

5.3. Constructors.

- *Sha1* → *(none)*
  The *Sha1* constructor creates a default hashing object that implements the SHA-1 algorithm.
- *Sha1* → *(Integer)*
  The *Sha1* constructor creates a SHA-1 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

6. Object Sha224

The *Sha224* class is a hashing class that implements the SHA-224 algorithm.

6.1. Predicate.

- sha224-p

6.2. Inheritance.

- Hasher

6.3. Constructors.

- *Sha224* → *(none)*
  The *Sha224* constructor creates a default hashing object that implements the SHA-224 algorithm.
- *Sha224* → *(Integer)*
  The *Sha224* constructor creates a SHA-224 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

7. Object Sha256

The *Sha256* class is a hashing class that implements the SHA-256 algorithm.

7.1. Predicate.

- sha256-p

7.2. Inheritance.

- Hasher

7.3. Constructors.

- *Sha256* → *(none)*
  The *Sha256* constructor creates a default hashing object that implements the SHA-256 algorithm.
- *Sha256* → *(Integer)*
  The *Sha256* constructor creates a SHA-256 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

8. Object Sha384

The *Sha384* class is a hashing class that implements the SHA-384 algorithm.

8.1. Predicate.

- sha384-p

8.2. Inheritance.

- Hasher

8.3. Constructors.

- *Sha384* → *(none)*
  The *Sha384* constructor creates a default hashing object that implements the SHA-384 algorithm.
- *Sha384* → *(Integer)*
  The *Sha384* constructor creates a SHA-384 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

9. Object Sha512

The *Sha512* class is a hashing class that implements the SHA-512 algorithm.

9.1. Predicate.

- sha512-p

9.2. Inheritance.

- Hasher

9.3. Constructors.

- *Sha512* → *(none)*
  The *Sha512* constructor creates a default hashing object that implements the SHA-512 algorithm.
- *Sha512* → *(Integer)*
  The *Sha512* constructor creates a SHA-512 hashing object with a result length. The argument is the hasher result length that must be less or equal to the hasher length.

10. Object Key

The *Key* class is an original class used to store a particular key or to generate one. A key is designed to operate with a variety of cipher that can be either symmetric or asymmetric. In the symmetric case, the key is generally an array of bytes. Asymmetric key are generally stored in the form of number list that can be computed or loaded by value. By default, a random 128 bit symmetric key is created.

10.1. Predicate.

- key-p

10.2. Inheritance.

- Object

10.3. Constructors.

- *Key → (none)*
  The *Key* constructor creates a default cipher key. The key is generated with random bytes and is 128 bits long.
- *Key → (String)*
  The *Key* constructor creates a symmetric key from an octet string. The octet string argument determines the size of the key. The octet string argument is compatible with the string obtained from the *format* method.
- *Key → (Item)*
  The *Key* constructor creates a key by type. If the key type is *KSYM* , a symmetric 128 bytes random key is generated. If the key type is *KRSA* , a 1024 bits RSA random key is generated.
- *Key → (Item Integer—String—Vector)*
  The *Key* constructor creates a key by type. The first argument is the key type to generate. The second argument is either the key size, the key octet string or the key byte values. In the first form, an integer argument specifies the key size in bytes or bits depending on the key nature. In the second form, a string is used as octet string to represent the key. In the third form, a vector of byte values can be used to load the key.

10.4. Constants.

- *KSYM → ()*
  The *KSYM* constant indicates that the key is a symmetric key.
- *KRSA → ()*
  The *KRSA* constant indicates that the key is a asymmetric RSA key.
- *KMAC → ()*
  The *KMAC* constant indicates that the key is a message authentication (MAC) key.
- *RSA-MODULUS → ()*
  The *RSA-MODULUS* constant corresponds to the RSA modulus value.
- *RSA-PUBLIC-EXPONENT → ()*
  The *RSA-PUBLIC-EXPONENT* constant corresponds to the RSA public exponent value which is generally 65537.
- *RSA-SECRET-EXPONENT → ()*
  The *RSA-SECRET-EXPONENT* constant corresponds to the RSA secret exponent value.

10.5. Methods.

- *get-byte → Byte (Integer)*
  The *get-byte* method returns a key byte value by index. The index must be in the key range or an exception is raised. This method is primarily used with symmetric key.
- *get-type → Item (none)*
  The *get-type* method returns the key type in the form of an item object.
- *get-bits → Integer (none)*
  The *get-bits* method returns the key size in bits.
- *get-size → Integer (none)*
  The *get-size* method returns the key size in bytes.
- *format → String (none—Item)*
  The *format* method returns a string representation of the key. In the first form, without argument, the key is returned as an octet string if possible. In the second form, the key value is returned as an octet string based on the key element to access.
- *get-relatif-key → Relatif (Item)*
  The *get-relatif-key* method returns a relatif representation of a key element. This method is well suited for asymmetric key. The key value is returned as a relatif based on the key element to access.

11. Object Kdf

The *Kdf* class is an abstract class used to model key derivation function. The class provides only a byte buffer which can be accessed by index. In the key derivation functions land, there are numerous standards, such like PKCS 2.1, IEEE P1363-2000, ISO/IEC 18033-2. All of these standards have sometimes conflicting definitions.

11.1. Predicate.

- kdf-p

11.2. Inheritance.

- Nameable

11.3. Methods.

- *reset → none (none)*
  The *reset* method reset the internal state of the kdf object.
- *get-size → Integer (none)*
  The *get-size* method returns the kdf size in bytes.
- *get-byte → Byte (Integer)*
  The *get-byte* method returns a kdf byte value by index. The index must be in the key range or an exception is raised.
- *format → String (none)*
  The *format* method returns a string representation of the derived key.
- *derive → String (String)*
  The *derive* method returns a string representation of a derived key computed from the octet string argument.
- *compute → String (String)*
  The *compute* method returns a string representation of a derived key computed from the string argument.

12. Object Hkdf

The *Hkdf* class is an abstract class used to model key derivation function based on hash function. The class maintains a hasher object that is used to derive the key from an octet string.

12.1. Predicate.

- hashed-kdf-p

12.2. Inheritance.

- Kdf

12.3. Methods.

- *get-hasher* → *none (none)*
  The *get-hasher* method returns the hasher object associated with the key derivation function object. object.

13. Object Kdf1

The *Kdf1* class is a hashed key derivation function class that implements the KDF1 specification as defined by ISO/IEC 18033-2. The class is strictly equivalent to the mask generation function (MGF1) defined in PKCS 2.1. On the other hand, this implementation does not conform to the KDF1 specification of IEEE 1363-2000 which is somehow rather bizarre. The class operates in theory with any type of hasher object as long as the octet string is not too long.

13.1. Predicate.

- kdf1-p

13.2. Inheritance.

- Hkdf

13.3. Constructors.

- *Kdf1* → *(Hasher Integer)*
  The *Kdf1* constructor creates a KDF1 key derivation function object. The first argument is the hasher object to bind and the second argument is the kdf size.

14. Object Kdf2

The *Kdf2* class is a hashed key derivation function class that implements the KDF2 specification as defined by ISO/IEC 18033-2. The class is strictly equivalent to the key function derivation (KDF1) except that the internal counter runs from 1 to k instead of 0 to k-1. The class operates in theory with any type of hasher object as long as the octet string is not too long.

14.1. Predicate.

- kdf2-p

14.2. Inheritance.

- Hkdf

14.3. Constructors.

- *Kdf2* → *(Hasher Integer)*
  The *Kdf2* constructor creates a KDF2 key derivation function object. The first argument is the hasher object to bind and the second argument is the kdf size.

15. Object Cipher

The *Cipher* class is a base class that is used to implement a cipher. A cipher is used to encrypt or decrypt a message. There are basically two types of ciphers, namely symmetric cipher and asymmetric cipher. For the base class operation, only the cipher name and key is needed. A reverse flag controls whether or not an encryption operation must be reversed. A reset method can also be used to reset the internal cipher state.

15.1. Predicate.

- cipher-p

15.2. Inheritance.

- Nameable

15.3. Methods.

- *reset → none (none)*
  The *reset* method reset the cipher internal state.
- *stream → Integer (OutputStream InputStream)*
  The *stream* method process an input stream and write into an output stream. The method returns the number of character processed. The first argument is the output stream used to write the coded characters. The second argument is the input stream used to read the characters.
- *set-key → none (Key)*
  The *set-key* method sets the cipher key. The first argument is the key to set.
- *get-key → Key (none)*
  The *get-key* method returns the cipher key.
- *set-reverse → none (Boolean)*
  The *set-reverse* method sets the cipher reverse flag. The first argument is the flag to set. If the flag is true, the cipher operates in reverse mode. If the flag is false, the cipher operates in direct mode.
- *get-reverse → Boolean (none)*
  The *get-reverse* method returns the cipher reverse flag. If the flag is true, the cipher operates in reverse mode. If the flag is false, the cipher operates in direct mode.

16. Object BlockCipher

The *BlockCipher* class is an abstract class that is used to implement a symmetric block cipher. By default the cipher operates in encryption mode. When the reverse flag is set, the decryption mode is activated. For a block cipher, a block size controls the cipher operations. The class also defines the constants that control the block padding with the associated methods.

16.1. Predicate.

- block-cipher-p

16.2. Inheritance.

- Cipher

16.3. Constants.

- *PAD-NONE → ()*
  The *PAD-NONE* constant indicates that the block should not be padded.
- *PAD-BIT-MODE → ()*
  The *PAD-BIT* constant indicates that the block should be padded in bit mode.
- *PAD-ANSI-X923 → ()*
  The *PAD-ANSI-X923* constant indicates that the block should be padded according to ANSI X 923 standard.
- *PAD-NIST-800 → ()*
  The *PAD-NIST-800* constant indicates that the block should be padded according to NIST 800-38A recommendations. This is the default mode.

16.4. Methods.

- *get-block-size → Integer (none)*
  The *get-block-size* method returns the cipher block size.
- *set-padding-mode → none (Item)*
  The *set-padding-mode* method sets the cipher padding mode.
- *get-padding-mode → Item (none)*
  The *get-padding-mode* method returns the cipher padding mode.

17. Object InputCipher

The *InputCipher* class is an stream interface that can stream out an input stream from a cipher. In other word, an input stream is read and block are encoded as long as the input stream read characters. If the cipher is nil, the input cipher simply read the input stream and is therefore transparent. The class acts like an input stream, read the character from the bounded input stream and encode or decode them from the bounded cipher. The *InputCipher* defines several modes of operations. In *electronic codebook mode* or ECB, the character are encoded in a block basis. In *cipher block chaining* mode, the block are encoded by doing an XOR operation with the previous block. Other modes such like *cipher feedback mode* and *output feedback mode* are also defined.

17.1. Predicate.

- input-cipher-p

17.2. Inheritance.

- Input

17.3. Constructors.

- *InputCipher → (Cipher)*
  The *InputCipher* constructor creates an input cipher with a cipher object. The first argument is the cipher to used for processing.
- *InputCipher → (Cipher Input)*
  The *InputCipher* constructor creates an input cipher with a cipher object and an input stream. The first argument is the cipher to used for processing. The second argument is the input stream object used for the character reading.
- *InputCipher → (Cipher InputStream Item)*
  The *InputCipher* constructor creates an input cipher with a cipher object, an input stream and a mode. The first argument is the cipher to used for processing. The second argument is the input stream object used for the character reading. The third argument is the input cipher mode which can be either ECB, CBC, CFB or OFB.

17.4. Constants.

- *ECB → ()*
  The *ECB* constant indicates that the input cipher is to operate in *electronic codebook* mode. This mode is the default mode.
- *CBC → ()*
  The *CBC* constant indicates that the input cipher is to operate in *cipher chaining block* mode.
- *CFB → ()*
  The *CFB* constant indicates that the input cipher is to operate in *cipher feedback block* mode.
- *OFB → ()*
  The *OFB* constant indicates that the input cipher is to operate in *output feedback block* mode.

17.5. Methods.

- *reset → none (none)*
  The *reset* method reset the input cipher object.
- *get-mode → Item (none)*
  The *get-mode* method returns the input cipher operating mode.

- *set-iv → none (String—Buffer)*
  The *set-iv* method sets the input cipher initial vector. In the first form, the initial vector is set from an octet string. In the second form, the initial vector is set from a buffer object.
- *get-iv → String (none)*
  The *get-iv* method returns the input cipher initial vector as an octet string.
- *set-is → none (InputStream)*
  The *set-is* method sets the input cipher input stream. This method can be used to chain multiple input streams in a unique coding session.

18. Object Aes

The *Aes* class is a block cipher class that implements the *advanced encryption standard* (AES), originally known as Rijndael. This is an original implementation that conforms to the standard FIPS PUB 197. It should be noted that the AES standard, unlike Rijndael, defines a fixed block size of 16 bytes (4 words) and 3 keys sizes (128, 192, 256).

18.1. Predicate.

- aes-p

18.2. Inheritance.

- BlockCipher

18.3. Constructors.

- *Aes → (Key)*
  The *Aes* constructor creates a direct cipher with a key. The first argument is the key used by the cipher.
- *Aes → (Key Boolean)*
  The *Aes* constructor creates a cipher with a key and a reverse flag. The first argument is the key used by the cipher. The second argument is the reverse flag.

19. Object PublicCipher

The *PublicCipher* class is an abstract class that is used to implement an asymmetric cipher. An asymmetric cipher or public key cipher is designed to operate with a public key and a secret key. Depending on the use model, the public key might be used to crypt the data, and the secret key to decrypt. The basic assumption around a public cipher is that the secret key cannot be derived from the public key.

19.1. Predicate.

- public-cipher-p

19.2. Inheritance.

- Cipher

19.3. Methods.

- *get-message-size → Integer (none)*
  The *get-message-size* method returns the cipher message size.
- *get-crypted-size → Integer (none)*
  The *get-crypted-size* method returns the cipher crypted block size.

20. Object Rsa

The *Rsa* class is a public cipher class that implements the RSA algorithm as described by PKCS 2.1, RFC 2437 and ISO 18033-2. The class implements also some padding mechanism described in PKCS 1.5, 2.1 and ISO 18033-2. The RSA algorithm is a public cryptographic cipher based on a secret and public keys. The class operates in crypting mode by default and uses the public key to do the encryption while the secret key is used in reverse (decryption) mode. By default, the PKCS 1.5 type 2 padding is used. The ISO RSA-REM1 padding with a key derivation function (KDF1) is equivalent to PKCS 2.1 padding with the mask generation function (MGF1). The ISO RSA-REM1 padding with KDF2 is not described in the PKCS 2.1.

20.1. Predicate.

- rsa-p

20.2. Inheritance.

- PublicCipher

20.3. Constructors.

- *Rsa → (none)*
  The *Rsa* constructor creates a default RSA public cipher by binding a 1024 bits random key.
- *Rsa → (Key)*
  The *Rsa* constructor creates a RSA public cipher by binding the key argument.
- *Rsa → (Key Boolean)*
  The *Rsa* constructor creates a RSA public cipher by binding the key argument and the reverse flag. The first argument is the key to bind. The second argument is the reverse flag to set.
- *Rsa → (Key Hasher String)*
  The *Rsa* constructor creates a RSA public cipher by binding the key argument and OAEP padding objects. The first argument is the key to bind. The second argument is hasher object to use with the OAEP padding mode. The third argument is an optional label to be used by the KDF object.

20.4. Constants.

- *PAD-PKCS-11 → ()*
  The *PAD-PKCS-11* constant indicates that the PKCS 1.5 type 1 block should be used to pad the message.
- *PAD-PKCS-12 → ()*
  The *PAD-PKCS-12* constant indicates that the PKCS 1.5 type 3 block should be used to pad the message.
- *PAD-OAEP-K1 → ()*
  The *PAD-OAEP-K1* constant indicates that the ISO/IEC 18033-2 OAEP with KDF1 should be used to pad the message.
- *PAD-OAEP-K2 → ()*
  The *PAD-OAEP-K2* constant indicates that the ISO/IEC 18033-2 OAEP with KDF2 should be used to pad the message.

20.5. Methods.

- *get-hasher → Hasher (none)*
  The *get-hasher* method returns the hasher object used by the OAEP padding mode.

- *set-hasher → none (Hasher)*
  The *set-hasher* method sets the hasher object used by the OAEP padding mode.
- *get-padding-mode → Item (none)*
  The *get-padding-mode* method returns the cipher padding mode.
- *set-padding-mode → none (Item)*
  The *set-padding-mode* method sets the cipher padding mode.
- *get-padding-label → String (none)*
  The *get-padding-label* method returns the cipher padding label.
- *set-padding-label → none (String)*
  The *set-padding-mode* method sets the cipher padding label.
- *get-padding-seed → String (none)*
  The *get-padding-seed* method returns the cipher padding seed.
- *set-padding-seed → none (String)*
  The *set-padding-seed* method sets the cipher padding seed.
- *pkcs-primitive → Relatif (Integer—Relatif)*
  The *pkcs-primitive* method compute a relatif value from a relatif argument by either crypting or decrypting the argument. seed.

21. Object Signer

The *Signer* class is a base class that is used to build a message signature. The signature result is stored in a special signature object which is algorithm dependent.

21.1. Predicate.

- signer-p

21.2. Inheritance.

- Nameable

21.3. Methods.

- *reset → none (none)*
  The *reset* method reset the signer object with its associated internal states.
- *compute → Signature (Literal—Buffer—InputStream)*
  The *compute* method computes the signature from a string, a buffer or an input stream. The method returns a signature object. When the argument is a buffer object or an input stream, the characters are consumed from the object.
- *derive → Signature (String)*
  The *derive* method computes the signature from an octet string which is converted before the signature computation. The method returns a signature object.

22. Object Signature

The *Signature* class is a container class designed to store a message signature. The signature object is produced by a signing process, implemented in the form of a digital signature algorithm such like RSA or DSA.

22.1. Predicate.

- signature-p

22.2. Inheritance.

- Object

22.3. Constructors.

- *Signature → (none)*
  The *Signature* constructor creates an empty signature.

22.4. Constants.

- *NIL → ()*
  The *NIL* constant indicates that the signature is a null signature.
- *DSA → ()*
  The *DSA* constant indicates that the signature is conforming to DSS.
- *DSA-S-COMPONENT → ()*
  The *DSA-S-COMPONENT* constant corresponds to the DSA S component value.
- *DSA-R-COMPONENT → ()*
  The *DSA-R-COMPONENT* constant corresponds to the DSA R component value.

22.5. Methods.

- *reset → none (none)*
  The *reset* method reset the signature object to a null signature.
- *format → String (Item)*
  The *format* method returns a string representation of the signature component. The signature component is returned as an octet string based on the signature component to access.
- *get-relatif-component → Relatif (Item)*
  The *get-relatif-component* method returns a relatif representation of a signature component.

23. Object Dsa

The *Dsa* class is an original implementation of the Digital Signature Standard (DSS) as published in FIPS PUB 186-3. This class implements the Digital Signature Algorithm (DSA) with an approved key length of 1024, 2048 and 3072 bits with a 160, 224 and 256 bits hash function which is part of the SHA family.

23.1. Predicate.

- dsa-p

23.2. Inheritance.

- Signer

23.3. Constructors.

- *Dsa → (none)*
  The *Dsa* constructor creates a signer object with a default DSA key.
- *Dsa → (Key)*
  The *Dsa* constructor creates a signer object with a DSA key as its argument.
- *Dsa → (Key Relatif)*
  The *Dsa* constructor creates a signer object with a DSA key as its first argument and a fixed k argument as specified by DSS.

CHAPTER 13

Standard Input/Output Module

The *Standard Input/Output* module is an orginal implementation that provides objects for i/o operations. Although input and output files are the standard objects that one might expect, the module facilities for directory access, path manipulation and i/o event management. At the heart of this module is the concept of stream associated with the transcoding object which enable the passage between one coding system to another. It is also this module which provides the stream selector object.

1. Input and output streams

The *afnix-sio* module is based on facilities provided by two base classes, namely, the *InputStream* stream and the *OutputStream* stream. Both classes have associated predicates with the name *input-stream-p* and *output-stream-p* . The base class associated is the *Stream* class those sole purpose is to define the stream coding mode.

1.1. Stream base class. The *Stream* class is the base class for the *InputStream* and *OutputStream* classes. The *Stream* class is used to define the stream coding mode that affects how characters are read or written. When a stream operates in *byte mode* , each character is assumed to be encoded in one byte. In that case, the input stream methods *read* and *getu* are equivalent and no transformation is performed when writing characters. This behavior is the default stream behavior. For certain stream, like terminal, this behavior is changed depending on the current localization settings. For instance, if the current locale is operating with an *UTF-8* codeset, the *Terminal* stream coding mode is automatically adjusted to reflect this situation. Since the *US-ASCII* codeset is predominant and the default steam coding mode is the byte mode, there should be no conflict during the read and write operations.

1.2. Stream transcoding. The *Stream* class provides the support for the transcoding of different codesets. All *ISO-8859* codesets are supported. Since the engine operates internally with Unicode characters, the transcoding operation takes care of changing a character in one particular codeset into its equivalent Unicode representation. This operation is done for an input stream that operates in byte mode. For an output stream, the opposite operation is done. An internal Unicode characters representation is therefore mapped into a particular codeset. Note that only the codeset characters can be mapped.

The *set-encoding-mode* can be used to set the stream encoding codeset. The method operates either by enumeration or string. The *get-encoding-mode* returns the stream encoding mode. There are some time good reasons to force a stream encoding mode. For example, a file encoded in UTF-8 that is read will require this call since the default stream mode is to work in byte mode. It should be noted that there is a difference between the enumeration and the string encoding mode. The enumeration mode defines whether the stream operates in byte or UTF-8 mode. When the stream operates in byte mode, it is also necessary to define the transcoding mode with the *set-transcoding-mode* method. For simplicity, the string version of the *set-encoding-mode* takes care of setting both the stream

Codeset	Description
DEFAULT	Default codeset, i.e US-ASCII
ISO-01	ISO-8859-1 codeset
ISO-02	ISO-8859-2 codeset
ISO-03	ISO-8859-3 codeset
ISO-04	ISO-8859-4 codeset
ISO-05	ISO-8859-5 codeset
ISO-06	ISO-8859-6 codeset
ISO-07	ISO-8859-7 codeset
ISO-08	ISO-8859-8 codeset
ISO-09	ISO-8859-9 codeset
ISO-10	ISO-8859-10 codeset
ISO-11	ISO-8859-11 codeset
ISO-13	ISO-8859-13 codeset
ISO-14	ISO-8859-14 codeset
ISO-15	ISO-8859-15 codeset
ISO-16	ISO-8859-16 codeset
UTF-08	Unicode UTF-8 codeset

mode and the transcoding mode. It is also worth to note that internally, the *Stream* class is derived from the *Transcoder* class.

1.3. Input stream. The *InputStream* base class has several method for reading and testing for byte availability. Moreover, the class provides a push-back buffer. Reading bytes is in the form of three methods. The *read* method without argument returns the next available byte or the `end-of-stream` *eos* . With an integer argument, the *read* method returns a *Buffer* with at most the number of requested bytes. The *readln* method returns the next available line. When it is necessary to read characters instead of bytes, the *getu* is more appropriate since it returns an Unicode character.

1.4. Output stream. The *OutputStream* base class provides the base methods to write to an output stream. The *write* method takes literal objects which are automatically converted to string representation and then written to the output stream. Note that for the case of a *Buffer* object, it is the buffer itself that take a stream argument and not the opposite.

1.5. The valid-p predicate. The input stream provides a general mechanism to test and read for bytes. The base method is the *valid-p* predicate that returns *true* if a byte can be read from the stream. It is important to understand its behavior which depends on the stream type. Without argument, the *valid-p* predicate checks for an available byte from the input stream. This predicate will block if no byte is available. On the other end, for a bounded stream like an input file, the method will not block at the end of file. With one integer argument, the *valid-p* predicate will timeout after the specified time specified in milliseconds. This second behavior is particularly useful with unbound stream like socket stream.

1.6. The eos-p predicate. The *eos-p* predicate does not take argument. The predicate behaves like *not (valid-p 0)* . However, there are more subtle behaviors. For an input file, the predicate will return *true* if and only if a byte cannot be read. If a byte has been pushed-back and the `end-of-stream` marker is reached, the method will return false. For an input terminal, the method returns true if the user and entered the `end-of-stream` byte. Once again, the method reacts to the contents of the push-back buffer. For certain input

stream, like a tcp socket, the method will return true when no byte can be read, that is here, the connection has been closed. For an udp socket, the method will return *true* when all datagram bytes have be read.

1.7. The read method. The *read* method is sometimes disturbing. Nevertheless, the method is a blocking one and will return a byte when completed. The noticeable exception is the returned byte when an `end-of-stream` marker has been reached. The method returns the `ctrl-d` byte. Since a binary file might contains valid byte like `ctrl-d` it is necessary to use the *valid-p* or *eos-p* predicate to check for a file reading completion. This remark apply also to bounded streams like a tcp socket. For some type of streams like a udp socket, the method will block when all datagram bytes have been consumed and no more datagram has arrived. With this kind of stream, there is no `end-of-stream` condition and therefore care should be taken to properly assert the stream content. This last remark is especially true for the *readln* method. The method will return when the `end-of-stream` marker is reached, even if a newline byte has not been read. With an udp socket, such behavior will not happen.

1.8. Buffer read mode. The *read* method with an integer argument, returns a buffer with at least the number of bytes specified as an argument. This method is particularly useful when the contents has a precise size. The method returns a *Buffer* object which can later be used to read, or transform bytes. Multi-byte conversion to number should use such approach. The *read* method does not necessarily returns the number of requested bytes. Once the buffer is returned, the *length* method can be used to check the buffer size. Note also the existence of the *to-string* method which returns a string representation of the buffer.

```
# try to read 256 bytes
const buf (is:read 256)
# get the buffer size
println (buf:length)
# get a string representation
println (buf:to-string)
```

2. File stream

The *afnix-sio* module provides two classes for file access. The *InputFile* class open a file for input. The *OutputFile* class opens a file for output. The *InputFile* class is derived from the *InputStream* base class. The *OutputFile* class is derived from the *OutputStream* class. By default an output file is created if it does not exist. If the file already exist, the file is truncated to 0. Another constructor for the output file gives more control about this behavior. It takes two boolean flags that defines the truncate and append mode.

```
# load the module
interp:library "afnix-sio"
# create an input file by name
const if (afnix:sio:InputFile "orig.txt")
# create an output file by name
const of (afnix:sio:OutputFile "copy.txt")
```

2.1. Stream information. Both *InputFile* and *OutputFile* supports the *get-name* method which returns the file name.

```
println (if:get-name)
println (of:get-name)
```

Predicates are also available for these classes. The *input-file-p* returns true for an input file object.The *output-file-p* returns true for an output file object.

```
afnix:sio:input-stream-p if
afnix:sio:output-stream-p of
afnix:sio:input-file-p if
afnix:sio:output-file-p of
```

2.2. Reading and writing. The *read* method reads a byte on an input stream. The *write* method writes one or more literal arguments on the output stream. The *writeln* method writes one or more literal arguments followed by a newline byte on the output stream. The *newline* method write a newline byte on the output stream. The *eos-p* predicate returns true for an input stream, if the stream is at the end. The *valid-p* predicate returns true if an input stream is in a valid state. With these methods, copying a file is a simple operation.

```
# load the module and open the files
interp:library "afnix-sio"
const if (afnix:sio:InputFile "orig.txt")
const of (afnix:sio:OutputFile "copy.txt")
# loop in the input file and write
while (if:valid-p) (of:write (if:read))
```

The use of the *readln* method can be more effective. The example below is a simple cat program which take the file name an argument.

```
# cat a file on the output terminal
# usage: axi 0601.als file
# get the io module
interp:library "afnix-sio"
# cat a file
const cat (name) {
  const f (afnix:sio:InputFile name)
  while (f:valid-p) (println (f:readln))
  f:close
}
# get the file
if (== 0 (interp:argv:length)) {
  errorln "usage: axi 0601.als file"
} {
  cat (interp:argv:get 0)
}
```

3. Multiplexing

I/O multiplexing is the ability to manipulate several streams at the same time and process one at a time. Although the use of threads reduce the needs for i/o multiplexing, there is still situations where they are needed. In other words, I/O multiplexing is identical to the *valid-p* predicate, except that it works with several stream objects.

3.1. Selector object. I/O multiplexing is accomplished with the *Selector* class. The constructor takes 0 or several stream arguments. The class manages automatically to differentiate between *InputStream* stream and *OutputStream* streams. Once the class is constructed, it is possible to get the first stream ready for reading or writing or all of them. We assume in the following example that *is* and *os* are respectively an input and an output stream.

```
# create a selector
const slt (afnix:sio:Selector is)
# at this stage the selector has one stream
# the add method can add more streams
slt:add os
```

The *add* method adds a new stream to the selector. The stream must be either an *InputStream* and *OutputStream* stream or an exception is raised. If the stream is both an input and an output stream, the preference is given to the input stream. If this preference is not acceptable, the *input-add* or the *output-add* methods might be preferable. The *input-length* method returns the number of input streams in this selector. The *output-length* method returns the number of output streams in this selector. The *input-get* method returns the selector input stream by index. The *output-get* method returns the selector output stream by index.

3.2. Waiting for i/o event. The *wait* and *wait-all* methods can be used to detect a status change in the selector. Without argument both methods will block indefinitely until one stream change. With one integer argument, both method blocks until one stream change or the integer argument timeout expires. The timeout is expressed in milliseconds. Note that 0 indicates an immediate return. The *wait* method returns the first stream which is ready either for reading or writing depending whether it is an input or output stream. The *wait-all* method returns a vector with all streams that have changed their status. The *wait* method returns *nil* if the no stream have changed. Similarly, the *wait-all* method returns an empty vector.

```
# wait for a status change
const is (slt:wait)
# is is ready for reading - make sure it is an input one
if (afnix:sio:input-stream-p is) (is:read)
```

A call to the *wait* method will always returns the first input stream.

3.3. Marking mode. When used with several input streams in a multi-threaded context, the selector behavior can becomes quite complicated. For this reason, the selector can be configured to operate in marking mode. In such mode, the selector can be marked as ready by a thread independently of the bounded streams. This is a useful mechanism which can be used to cancel a select loop. The *mark* method is designed to mark the selector while the *marked-p* predicate returns true if the stream has been marked.

4. Terminal streams

Terminal streams are another kind of streams available in the standard i/o module. The *InputTerm* , *OutputTerm* and *ErrorTerm* classes are low level classes used to read or write from or to the standard streams. The basic methods to read or write are the same as the file streams. Reading from the input terminal is not a good idea, since the class does not provide any formatting capability. One may prefer to use the *Terminal* class. The use of the output terminal or error terminal streams is convenient when the interpreter standard streams have been changed but one still need to print to the terminal.

4.1. Terminal class. The *Terminal* class combines an input stream and an output stream with some line editing capabilities. When the class is created, the constructed attempts to detect if the input and output streams are bounded to a terminal (i.e tty). If the line editing capabilities can be loaded (i.e non canonical mode), the terminal is initialized for line editing. Arrows, backspace, delete and other control sequences are available when using the *read-line* method. The standard methods like *read* or *readln* do not use the line editing features. When using a terminal, the prompt can be set to whatever the user wishes with the methods *set-primary-prompt* or *set-secondary-prompt* . A secondary prompt is displayed when the *read-line* method is called with the boolean argument false.

```
const term (Terminal)
term:set-primary-prompt "demo:"
const line (term:read-line)
errorln line
```

4.2. Using the error terminal. The *ErrorTerm* class is the most frequently used class for printing data on the standard error stream. The reserved keywords *error* or *errorln* are available to write on the interpreter error stream. If the interpreter error stream has been changed, the use of the *ErrorTerm* will provide the facility required to print directly on the terminal. The *cat* program can be rewritten to do exactly this.

```
# cat a file on the error terminal
# get the io module
interp:library "afnix-sio"
# cat a file
const cat (name es) {
  const f (afnix:sio:InputFile name)
  while (f:valid-p) (es:writeln (f:readln))
  f:close
}
```

5. Directory

The *Directory* class provides a facility to manipulate directories. A directory object is created either by name or without argument by considering the current working directory. Once the directory object is created, it is possible to retrieve its contents, create new directory or remove empty one.

5.1. Reading a directory. A *Directory* object is created either by name or without argument. With no argument, the current directory is opened. When the current directory is opened, its full name is computed internally and can be retrieved with the *get-name* method.

```
# print the current directory
const pwd (afnix:sio:Directory)
println (pwd:get-name)
```

Once the directory object is opened, it is possible to list its contents. The *get-list* method returns the full contents of the directory object. The *get-files* method returns a list of files in this directory. The *get-subdirs* method returns a list of sub directories in this directory.

```
# print a list of files
const pwd (afnix:sio:Directory)
const lsf (d:get-files)
for (name) (lsf) (println name)
```

5.2. Creating and removing directories. The *mkdir* and *rmdir* methods can be used to create or remove a directory. Both methods take a string argument and construct a full path name from the directory name and the argument. This approach has the advantage of being file system independent. If the directory already exists, the *mkdir* methods succeeds. The *rmdir* method requires the directory to be empty.

```
const tmp (afnix:sio:Directory (
   afnix:sio:absolute-path "tmp"))
const exp (tmp:mkdir "examples")
const lsf (exp:get-files)
println (lsf:length)
tmp:rmdir "examples"
```

The function *absolute-path* constructs an absolute path name from the argument list. If relative path needs to be constructed, the function *relative-path* might be used instead.

6. Logtee

The *Logtee* class is a message logger facility associated with an output stream. When a message is added to the logger object, the message is also sent to the output stream, depending on the controlling flags. The name "logtee" comes from the contraction of "logger" and "tee". One particularity of the class is that without a stream, the class behaves like a regular logger.

6.1. Creating a logger. The *Logtee* default constructor creates a standard logger object without an output stream. The instance can also be created by size or with an output stream or both. A third method can also attach an information string.

```
# create a logger with the interpreter stream
const log (Logtee (interp:get-output-stream))
assert true (logger-p log)
```

6.2. Adding messages. The process of adding messages is similar to the regular logger. The only difference is that the message is placed on the output stream if a control flag is set and the message level is less or equal the report level. In the other word, the control flag controls the message display – the tee operation – while the report level filters some of the messages.

```
log:add 2 "a level 2 message"
```

The *set-tee* method sets the control flag. The *set-report-level* method sets the report level. Note that the *set-report-level* and its associated *get-report-level* method is part of the base *Logger* class.

7. Path name

The *Pathname* class is a base class designed to ease the manipulation of system path. It is particularly useful when it come to manipulate directory component.

7.1. Creating a path name. A path name is created either by file name or by file and directory name. In the first case, only the file name is used. In the second case, the full path name is characterized.

```
# create a new path name
const path (afnix:sio:Pathname "axi")
```

7.2. Adding a directory path. The best way to add a directory path is to use the *absolute-path* or the *relative-path* functions.

```
# adding a directory path
const name (afnix:sio:absolute-path "usr" "bin")
path:set-directory-name name
```

7.3. Getting the path information. The path information can be obtained individually or globally. The *get-file-name* and *get-directory-name* methods return respectively the file and directory name. The *get-root* method returns the root component of the directory name. The *get-full* method returns the full path name.

CHAPTER 14

Standard Input/Output Reference

1. Object Transcoder

The *Transcoder* class is a codeset transcoder class. The class is responsible to map a byte character in a given codeset into its associated Unicode character. It should be noted that not all characters can be transcoded.

1.1. Predicate.

- transcoder-p

1.2. Inheritance.

- Object

1.3. Constants.

- *DEFAULT* → *()*
  The *DEFAULT* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode. In default mode, each character is not transcoded. This mode is the identity mode.
- *I8859-01* → *()*
  The *I8859-01* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-6 codeset.
- *I8859-02* → *()*
  The *I8859-02* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-2 codeset.
- *I8859-03* → *()*
  The *I8859-03* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-3 codeset.
- *I8859-04* → *()*
  The *I8859-04* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-4 codeset.
- *I8859-05* → *()*
  The *I8859-05* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-5 codeset.
- *I8859-06* → *()*
  The *I8859-06* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-6 codeset.
- *I8859-07* → *()*
  The *I8859-07* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-7 codeset.
- *I8859-08* → *()*
  The *I8859-08* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-8 codeset.
- *I8859-09* → *()*
  The *I8859-09* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-9 codeset.
- *I8859-10* → *()*
  The *I8859-10* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-10 codeset.
- *I8859-11* → *()*
  The *I8859-11* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-11 codeset.

- *I8859-13 → ()*
  The *I8859-13* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-13 codeset.
- *I8859-14 → ()*
  The *I8859-14* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-14 codeset.
- *I8859-15 → ()*
  The *I8859-15* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-15 codeset.
- *I8859-16 → ()*
  The *I8859-16* constant is used by the *set-transcoding-mode* method to specify the class transcoding mode that corresponds to the ISO-8859-16 codeset.

1.4. Constructors.

- *Transcoder → (none)*
  The *Transcoder* constructor creates a default transcoder that operates in default mode by using the identity function.
- *Transcoder → (constant)*
  The *Transcoder* constructor creates a transcoder with the argument mode.

1.5. Methods.

- *set-transcoding-mode → none (constant)*
  The *set-transcoding-mode* method sets the class transcoding mode.
- *get-transcoding-mode → constant (none)*
  The *get-transcoding-mode* method returns the class transcoding mode.
- *valid-p → Byte—Character (Boolean)*
  The *valid-p* predicate returns true if character can be transcoded. If the argument is a byte, the method returns true if the byte can be transcoded to a character. If the argument is a character, the method returns true if the character can be transcoded to a byte.
- *encode → Byte (Character)*
  The *encode* method encodes a byte into a character. If the character cannot be encoded, an exception is raised.
- *decode → Character (Byte)*
  The *decode* method decodes a character into a byte. If the character cannot be decoded, an exception is raised.

2. Object Stream

The *Stream* class is a base class for the standard streams. The class is automatically constructed by a derived class and provides the common methods for all streams.

2.1. Predicate.

- stream-p

2.2. Inheritance.

- Transcoder

2.3. Constants.

- *BYTE → ()*
  The *BYTE* constant is used by the *set-coding-mode* method to specify the stream coding mode. In byte mode, each character is assumed to be coded with one byte. This value affects the *getu* and *write* methods
- *UTF-8 → ()*
  The *UTF-8* constant is used by the *set-coding-mode* method to specify the stream coding mode. In UTF-8 mode, each character is assumed to be coded in the UTF-8 representation. This value affects the *getu* and *write* methods

2.4. Methods.

- *set-encoding-mode → none (constant—String)*
  The *set-encoding-mode* method sets the stream coding mode that affects how characters are read or written. In the enumeration form, the method only sets the stream coding mode which is either byte or UTF-8 mode. In the string mode, the method sets the stream encoding mode and the transcoding mode.
- *get-encoding-mode → constant (none)*
  The *get-coding-mode* method returns the stream coding mode which affects how characters are read or written.

3. Object InputStream

The *InputStream* class is a base class for the standard i/o module. The class is automatically constructed by a derived class and provides the common methods for all input streams. The input stream is associated with a timeout value which is used for read operation. By default, timeout is infinite, meaning that any read without data will be a blocking one.

3.1. Predicate.

- input-stream-p

3.2. Inheritance.

- Stream

3.3. Methods.

- *flush → none—Character (none)*
  The *flush* method the input stream buffer. In the first form, without argument, the input stream buffer is entirely flushed. In the second form, the input stream buffer is flushed until the character argument is found.
- *get-timeout → Integer (none)*
  The *get-timeout* method returns the input stream timeout. A negative value is a blocking timeout.
- *set-timeout → none (Integer)*
  The *set-timeout* method sets the input stream timeout. A negative value is a blocking timeout. Changing the stream timeout does not cancel any pending read operation.
- *read → Byte (none)*
  The *read* method returns the next byte available from the input stream. If the stream has been closed or consumed, the `end-of-stream` byte is returned.
- *read → Buffer (Integer)*
  The *read* method returns a buffer object with at most the number of bytes specified as an argument. The buffer *length* method should be used to check how many bytes have been placed in the buffer.
- *readln → String (none)*
  The *readln* method returns the next line available from the input stream. If the stream has been closed or consumed, the `end-of-stream` character is returned.
- *getu → Character (none)*
  The *getu* method returns the next available Unicode character from the input stream. If the stream has been closed or consumed, the `end-of-stream` character is returned. During the read process, if the character decoding operation fails, an exception is raised.
- *valid-p → Boolean (none—Integer)*
  The *valid-p* method returns true if the input stream is in a valid state. By valid state, we mean that the input stream can return a byte with a call to the read method. With one argument, the method timeout after the specified time in milliseconds. If the timeout is null, the method returns immediately. With -1, the method blocks indefinitely if no byte is available.
- *eos-p → Boolean (none)*
  The *eos-p* predicate returns true if the input stream has been closed or all bytes consumed.
- *pushback → Integer (Byte—Character—String)*
  The *pushback* method push-back a byte, an Unicode character or a string in the

input stream. Subsequent calls to read will return the last pushed bytes. Pushing a string is equivalent to push each encoded bytes of the string. The method returns the number of bytes pushed back.

- *consume → Integer (none)*
  The *consume* method consumes an input stream and places the read characters into the stream buffer. The method returns the number of consumed characters. This method is generally used in conjonction with the *to-string* method.
- *get-buffer-length → Integer (none)*
  The *get-buffer-length* method returns the length of the push-back buffer.
- *to-string → String (none)*
  The *to-string* method returns a string representation of the input stream buffer.

4. Object InputFile

The *InputFile* class provide the facility for an input file stream. An input file instance is created with a file name. If the file does not exist or cannot be opened, an exception is raised. The *InputFile* class is derived from the *InputStream* class.

4.1. Predicate.

- input-file-p

4.2. Inheritance.

- InputStreamNameable

4.3. Constructors.

- *InputFile → (String)*
  The *InputFile* constructor create an input file by name. If the file cannot be created, an exception is raised. The first argument is the file name to open.
- *InputFile → (String String)*
  The *InputFile* constructor create an input file by name and encoding mode. If the file cannot be created, an exception is raised. The first argument is the file name to open.The second argument is the encoding mode to use.

4.4. Methods.

- *close → Boolean (none)*
  The *close* method close the input file and returns true on success, false otherwise. In case of success, multiple calls return true.
- *lseek → none (Integer)*
  The *lseek* set the input file position to the integer argument. Note that the push-back buffer is reset after this call.
- *length → Integer (none)*
  The *length* method returns the length of the input file. The length is expressed in bytes.
- *get-modification-time → Integer (none)*
  The *get-modification-time* method returns the modification time of the file. The returned argument is suitable for the *Time* and *Date* system classes.

5. Object InputMapped

The *InputMapped* class is an input stream class that provides the facility for reading a mapped input stream. The input stream is mapped at construction given a file name, a size and a file offset. An anonymous mapped input stream can also be designed with a buffer object. Finally, without any information an always valid null input stream is constructed.

5.1. Predicate.

- input-mapped-p

5.2. Inheritance.

- InputStream

5.3. Constructors.

- *InputMapped → (none)*
  The *InputMapped* constructor create a null input stream. This stream acts as a null character generator.
- *InputMapped → (String—Buffer)*
  The *InputMapped* constructor create a mapped input stream by name or buffer. In the first form, a string is used as file name to be mapped an input stream. In the second form, a buffer is mapped as an input stream.
- *InputMapped → (String Integer Integer)*
  The *InputMapped* constructor create a mapped input stream by name, size and offset. The string argument is the file name to map. The second argument is the desired mapped size. The third argument is the offset inside the file before mapping it.

5.4. Methods.

- *lseek → none (Integer)*
  The *lseek* set the input mapped file position to the integer argument. Note that the push-back buffer is reset after this call.
- *length → Integer (none)*
  The *length* method returns the length of the input mapped file. The length is expressed in bytes.

6. Object InputString

The *InputString* class provide the facility for an input string stream. The class is initialized or set with a string and then behaves like a stream. This class is very useful to handle generic stream method without knowing what kind of stream is behind it.

6.1. Predicate.

- input-string-p

6.2. Inheritance.

- InputStream

6.3. Constructors.

- *InputString → (none)*
  The *InputString* constructor creates an empty input string.
- *InputString → (String)*
  The *InputString* constructor creates an input string by value.

6.4. Methods.

- *get → Byte (none)*
  The *get* method returns the next available byte from the input stream but do not remove it.
- *set → none (String)*
  The *set* method sets the input string by first resetting the push-back buffer and then initializing the input string with the argument value.

7. Object InputTerm

The *InputTerm* class provide the facility for an input terminal stream. The input terminal reads byte from the standard input stream. No line editing facility is provided with this class This is a low level class, and normally, the *Terminal* class should be used instead.

7.1. Predicate.

- input-term-p

7.2. Inheritance.

- InputStreamOutputStream

7.3. Constructors.

- *InputTerm → (none)*
  The *InputTerm* constructor creates a default input terminal.

7.4. Methods.

- *set-ignore-eos → none (Boolean)*
  The *set-ignore-eos* method set the input terminal `end-of-stream` ignore flag. When the flag is on, any character that match a `ctrl-d` is changed to the end-of-stream mapped character returned by a read. This method is useful to prevent a reader to exit when the `ctrl-d` byte is generated.
- *set-mapped-eos → none (Byte)*
  The *set-mapped-eos* method set the input terminal `end-of-stream` mapped character. By default the character is set to `end-of-line` . This method should be used in conjunction with the *set-ignore-eos* method.

8. Object OutputStream

The *OutputStream* class is a base class for the standard i/o module. The class is automatically constructed by a derived class and provide the common methods for all output streams.

8.1. Predicate.

- output-stream-p

8.2. Inheritance.

- Stream

8.3. Methods.

- *write → Integer (Literal+)*
  The *write* method write one or more literal arguments on the output stream. This method returns the number of characters written.
- *writeln → none (Literal+)*
  The *writeln* method write one or more literal argument to the output stream and finish with a newline. This method return nil.
- *errorln → none (Literal+)*
  The *errorln* method write one or more literal argument to the associated output error stream and finish with a newline. Most of the time, the output stream and error stream are the same except for an output terminal.
- *newline → none (none)*
  The *newline* method writes a new line byte to the output stream. The method returns nil.
- *write-soh → none (none)*
  The *write-soh* method writes a `start-of-heading` character to the output stream.
- *write-stx → none (none)*
  The *write-stx* method writes a `start-of-transmission` character to the output stream.
- *write-etx → none (none)*
  The *write-etx* method writes an `end-of-transmission` character to the output stream.
- *write-eos → none (none)*
  The *write-eos* method writes an `end-of-stream` character to the output stream.

9. Object OutputFile

The *OutputFile* class provide the facility for an output file stream. An output file instance is created with a file name. If the file does not exist, it is created. If the file cannot be created, an exception is raised. Once the file is created, it is possible to write literals. The class is derived from the *OutputStream* class. By default an output file is created if it does not exist. If the file already exist, the file is truncated to 0. Another constructor for the output file gives more control about this behavior. It takes two boolean flags that defines the truncate and append mode. The `t-flag` is the truncate flag. The `a-flag` is the append flag.

9.1. Predicate.

- output-file-p

9.2. Inheritance.

- OutputStreamNameable

9.3. Constructors.

- *OutputFile → (String)*
  The *OutputFile* constructor create an output file by name. If the file cannot be created, an exception is raised. The first argument is the file name to create.
- *OutputFile → (String String)*
  The *OutputFile* constructor create an output file by name and encoding mode. If the file cannot be created, an exception is raised. The first argument is the file name to create. The second argument is the encoding mode to use.
- *OutputFile → (String Boolean Boolean)*
  The *OutputFile* constructor create an output file by name. If the file cannot be created, an exception is raised. The first argument is the file name to create. The second argument is the truncate flag. If the file already exists and the truncate flag is set, the file is truncated to 0. The third argument is the append mode. If set to true, the file is open in append mode.

9.4. Methods.

- *close → Boolean (none)*
  The *close* method closes the output file and returns true on success, false otherwise. In case of success, multiple calls returns true.

10. Object OutputString

The *OutputString* class provide the facility for an output string stream. The class is initially empty and acts as a buffer which accumulate the write method bytes. The *to-string* method can be used to retrieve the buffer content.

10.1. Predicate.

- output-string-p

10.2. Inheritance.

- OutputStream

10.3. Constructors.

- *OutputString → (none)*
  The *OutputString* constructor creates a default output string.
- *OutputString → (String)*
  The *OutputString* constructor creates an output string by value. The output string stream is initialized with the string value.

10.4. Methods.

- *flush → none (none)*
  The *flush* method flushes the output stream by resetting the stream buffer.
- *length → Integer (none)*
  The *length* method returns the length of the output string buffer.
- *to-string → String (none)*
  The *to-string* method returns a string representation of the output string buffer.

11. Object OutputBuffer

The *OutputBuffer* class provide the facility for an output byte stream. The class is initially empty and acts as a buffer which accumulate the write method bytes. The *to-string* method can be used to retrieve the buffer content as a string. The *format* method can be used to retrieve the buffer content as an octet string. content.

11.1. Predicate.

- output-buffer-p

11.2. Inheritance.

- OutputStream

11.3. Constructors.

- *OutputBuffer → (none)*
  The *OutputBuffer* constructor creates a default output buffer.
- *OutputBuffer → (String)*
  The *OutputBuffer* constructor creates an output buffer by value. The output buffer stream is initialized with the string value.

11.4. Methods.

- *flush → none (none)*
  The *flush* method flushes the output stream by resetting the stream buffer.
- *length → Integer (none)*
  The *length* method returns the length of the output buffer.
- *to-string → String (none)*
  The *to-string* method returns a string representation of the output buffer.
- *format → String (none)*
  The *format* method returns an octet string representation of the output buffer.

12. Object OutputTerm

The *OutputTerm* class provide the facility for an output terminal. The output terminal is defined as the standard output stream. If the standard error stream needs to be used, the *ErrorTerm* class is more appropriate.

12.1. Predicate.

- output-term-p

12.2. Inheritance.

- OutputStream

12.3. Constructors.

- *OutputTerm* → *(none)*
  The *OutputTerm* constructor creates a default output terminal
- *ErrorTerm* → *(none)*
  The *ErrorTerm* constructor creates a default error terminal

13. Object Terminal

The *Terminal* class provides the facility for an i/o terminal with line editing capability. The class combines the *InputTerm* and *OutputTerm* methods.

13.1. Predicate.

- terminal-p

13.2. Inheritance.

- InputTermOutputTerm

13.3. Constructors.

- *Terminal → (none)*
  The *Terminal* constructor creates a default terminal which combines an input and output terminal with line editing capabilities.

13.4. Methods.

- *set-primary-prompt → none (String)*
  The *set-primary-prompt* method sets the terminal primary prompt which is used when the *read-line* method is called.
- *set-secondary-prompt → none (String)*
  The *set-secondary-prompt* method sets the terminal secondary prompt which is used when the *read-line* method is called.
- *get-primary-prompt → String (none)*
  The *get-primary-prompt* method returns the terminal primary prompt.
- *get-secondary → String (none)*
  The *get-secondary-prompt* method returns the terminal secondary prompt.

14. Object Intercom

The *Intercom* class is the interpreter communication class. The class operates with two streams. One output stream is used to send serialized data while the input stream is used to deserialize data. The *send* method can be used to send the data, while the *recv* can be used to receive them.

14.1. Predicate.

- intercom-p

14.2. Inheritance.

- Object

14.3. Constructors.

- *Intercom → (none)*
  The *Intercom* constructor creates a default interpreter communication object. There is no stream attached to it.
- *Intercom → (InputStream—OutputStream)*
  The *Intercom* constructor creates an interpreter communication object with an input or an output stream. In the first form, the input stream object is used by the *recv* method to read data object. In the second form, the output stream object is used by the *send* method to send data object.
- *Intercom → (InputStream OutputStream)*
  The *Intercom* constructor creates an interpreter communication object with an input and an output stream.

14.4. Methods.

- *send → none (Object)*
  The *send* method serialize the object argument with the help of the output stream bound to the interpreter communication object. If there is no output stream, nothing is sent.
- *recv → Object (none)*
  The *recv* method deserialize an object with the help of the input stream bound to the interpreter communication object. If there is no output stream, *nil* is returned.

- *request → Object (Object)*
  The *request* method perform an atomic send receive operation.
- *set-input-stream → none (InputStream)*
  The *set-input-stream* method binds an input stream to the interpreter communication object.
- *get-input-stream → InputStream (none)*
  The *get-input-stream* method returns the input stream bound to the interpreter communication object.
- *set-output-stream → none (OutputStream)*
  The *set-output-stream* method binds an output stream to the interpreter communication object.
- *get-output-stream → OutputStream (none)*
  The *get-output-stream* method returns the output stream bound to the interpreter communication object.

15. Object InputOutput

The *InputOutput* class implements an input-output stream with a buffer which holds character during the processing of transit between the output stream to the input stream. The theory of operation goes as follow. The internal buffer is filled with characters with the help of the output stream. The characters are consumed from the buffer with the help of the input stream (read method). If the buffer becomes empty the *eos-p* predicate returns true, the *valid-p* predicate false and the *read* method will return the *eos* character. The InputOutput buffer can also be initialized with a buffer. This provides a nice mechanism to use a buffer like an input stream. The i/o operations implemented by this class are non-blocking. As a consequence, it is not possible to suspend a thread with this class and have it awaken when some characters are available in the input stream.

15.1. Predicate.

- input-output-p

15.2. Inheritance.

- InputStreamOutputStream

15.3. Constructors.

- *InputOutput → (none)*
  The *InputOutput* constructor creates a default input/output stream.
- *InputOutput → (String)*
  The *InputOutput* constructor creates an input/output stream initialized with the string argument. The string argument is used to fill the string buffer.

15.4. Methods.

- *get → Byte (none)*
  The *get* method returns the next available byte from the input stream but do not remove it.
- *set → none (String)*
  The *set* method sets the input string by first resetting the push-back buffer and then initializing the input string with the argument value.

16. Object Selector

The *Selector* class provides some facilities to perform i/o multiplexing. The constructor takes 0 or several stream arguments.The class manages automatically the differentiation between the *InputStream* and the *OutputStream* objects. Once the class is constructed, it is possible to get the first stream ready for reading or writing or all of them. It is also possible to add more steams after construction with the *add* method. When a call to the *wait* method succeeds, the method returns the first available stream. If the *waitall* method is called, the method returns a vector with all ready steams. The selector can be configured to operate in marking mode. In such mode, the selector can be marked as ready by a thread independently of the bounded streams. This is a useful mechanism which can be used to cancel a select loop. The *mark* method is designed to mark the selector while the *marked-p* predicate returns true if the stream has been marked.

16.1. Predicate.

- selector

16.2. Inheritance.

- Object

16.3. Constructors.

- *Selector → (none)*
  The *Selector* constructor creates a default stream selector.
- *Selector → ([Boolean] [InputStream—OutputStream]*)*
  The *Selector* constructor creates a stream selector with 0 or more stream arguments. If the first argument is a boolean, the selector is constructed marked mode.

16.4. Methods.

- *add → none (InputStream—OutputStream)*
  The *add* method adds an input or output stream to the selector. If the stream is both an input and an output stream, the preference is given to the input stream. If this preference is not acceptable, the *input-add* or the *output-add* methods might be preferable.
- *input-add → none (InputStream)*
  The *input-add* method adds an input stream to the selector.
- *output-add → none (OutputStream)*
  The *output-add* method adds an output stream to the selector.
- *wait → Stream (none—Integer)*
  The *wait* method waits for a status change in the selector and returns the first stream that has change status. With one argument, the selector time-out after the specified time in milliseconds. Note that at the time of the return, several streams may have changed status.
- *wait-all → Vector (none—Integer)*
  The *wait* method waits for a status change in the selector and returns all streams that has change status in a vector object. With one argument, the selector time-out after the specified time in milliseconds. If the selector has timed-out, the vector is empty.
- *input-get → InputStream (Integer)*
  The *input-get* method returns the input streams in the selector by index. If the index is out of bound, an exception is raised.

- *output-get → OutputStream (Integer)*
  The *output-get* method returns the output streams in the selector by index. If the index is out of bound, an exception is raised.
- *input-length → Integer (none)*
  The *input-length* method returns the number of input streams in the selector.
- *output-length → Integer (none)*
  The *output-length* method returns the number of output streams in the selector.
- *mark → none (none)*
  The *mark* method marks a selector object.
- *marked-p → Boolean (none)*
  The *marked-p* predicate returns true if the selector has been marked.

17. Object Logtee

The *Logtee* class provides the facility of a logger object associated with an output stream. When a message is added, the message is written to the output stream depending on an internal flag. By default the tee mode is false and can be activated with the *set-tee* method.

17.1. Predicate.

- logtee-p

17.2. Inheritance.

- Logger

17.3. Constructors.

- *Logtee → (none)*
  The *Logtee* constructor creates a default logger without an output stream.
- *Logtee → (Integer)*
  The *Logtee* constructor creates a logger with a specific size without an output stream. terminal
- *Logtee → (OutputStream)*
  The *Logtee* constructor creates a logger with an output stream. The object is initialized to operate in write mode.
- *Logtee → (Integer OutputStream)*
  The *Logtee* constructor creates a logger with a specific size with an output stream. The first argument is the logger size. The second argument is the output stream.
- *Logtee → (Integer String OutputStream)*
  The *Logtee* constructor creates a logger with a specific size, an information string and an output stream. The first argument is the logger size. The second argument is information string. The third argument is the output stream.

17.4. Methods.

- *set-tee-stream → none (OutputStream)*
  The *set-tee-stream* method sets the tee output stream. This stream is different from the logger output stream
- *get-tee-stream → OutputStream (none)*
  The *get-tee-stream* method returns the object output stream.
- *set-tee → none (Boolean)*
  The *set-tee* method sets the object tee flag. When the flag is true, the logger writes the added message on the output stream.
- *get-tee → Boolean (none)*
  The *get-tee* method returns the object tee flag. When the flag is true, the logger writes the added message on the output stream.

18. Object Pathname

The *Pathname* class is a base class designed to manipulate system i/o paths. The class operates with a directory name and a file name. Both names are kept separated to ease the path manipulation. The path components can be extracted individually. However, it shall be noted that the first component has a special treatment to process the root directory name.

18.1. Predicate.

- pathname-p

18.2. Inheritance.

- Object

18.3. Constructors.

- *Pathname → (none)*
  The *Pathname* constructor creates a default path name without file and directory names.
- *Pathname → (String)*
  The *Pathname* constructor creates a path name with a file name. The first string argument is the file name.
- *Pathname → (String String)*
  The *Pathname* constructor creates a pathname with a file and directory name. The first string argument is the file name. The second string argument is the directory name.

18.4. Methods.

- *reset → none (none)*
  The *reset* method reset the path name by removing all path and file information.
- *dir-p → Boolean (none)*
  The *dir-p* predicate returns true if the path is a directory.
- *file-p → Boolean (none)*
  The *file-p* predicate returns true if the path is a file.
- *set-file-name → none (String)*
  The *set-file-name* method set the path name file name. The string argument is the file name.
- *get-file-name → String (none)*
  The *get-file-name* method returns the path name file name.
- *add-directory-name → none (String)*
  The *add-directory-name* method add the directory name to the directory path component. The string argument is the directory name.
- *set-directory-name → none (String)*
  The *set-directory-name* method set the directory name file name. The string argument is the directory name.
- *get-directory-name → String (none)*
  The *get-directory-name* method returns the path name directory name.
- *length → Integer (none)*
  The *length* method returns the number of directory path elements.
- *get-path → String (Integer)*
  The *get-path* method returns a directory path element by index.
- *get-root → String (none)*
  The *get-root* method returns the root component of a directory name.

- *get-full → String (none)*
  The *get-full* method returns the full path name by combining the directory name with the file name.
- *add-path → none (String)*
  The *add-path* method add a new path component by name. The path is separated into individual component and added to the directory path unless it is a root path. If the file name is set, the file name is added as a directory component. If the path is a root path, a new path name is rebuilt. This last case is equivalent to a call to *set-file-name* .
- *normalize → none (none)*
  The *normalize* method rebuild the path name by determining the full path nature if possible. In case of success, the path structure reflects the actual path type.

19. Object Pathlist

The *Pathlist* class is a base class designed to ease the manipulation of a file search path. The class acts like a list of search paths and various facilities are provided to find a valid path for a given name. The path list can be manipulated like any other list.

19.1. Predicate.

- pathlist-p

19.2. Inheritance.

- Object

19.3. Constructors.

- *Pathlist → (none)*
  The *Pathlist* constructor creates a default path list.
- *Pathlist → (Boolean—String)*
  The *Pathlist* constructor creates a path list with a local search flag or with an initial path component. In the first form, a boolean argument controls the local search flag. In the second for, a string argument is used as the initial path component.

19.4. Methods.

- *reset → none (none)*
  The *reset* method resets the path list by clearing the local search flag and removing all path components.
- *local-p → Boolean (none)*
  The *local-p* predicate returns true if the local search flag is set.
- *set-local-search → none (Boolean)*
  The *set-local-search* method sets the local search flag.
- *length → Integer (none)*
  The *length* method returns the number of directory path elements.
- *get-path → String (Integer)*
  The *get-path* method returns a directory path element by index.
- *add-path → none (String)*
  The *add-path* method add a new path component by name. The string argument is the name to add.
- *file-p → Boolean (String)*
  The *file-p* predicate returns true if the file name argument can be resolved. If the local search flag is set, the local directory is check first.
- *resolve → String (String)*
  The *resolve* method returns a string representation of the resolved file path. If the local search flag is set and the file name is found locally, the initial name argument is returned.

19.5. Functions.

- *dir-p → Boolean (String)*
  The *dir-p* function returns true if the argument name is a directory name, false otherwise.
- *file-p → Boolean (String)*
  The *file-p* function returns true if the argument name is a regular file name, false otherwise.

- *tmp-name → String (String?)*
  The *tmp-name* function returns a name suitable for the use as a temporary file name. Without argument, a default prefix is used to build the name. An optional string prefix can control the original name.
- *tmp-path → String (String?)*
  The *tmp-path* function returns a path suitable for the use as a temporary file name. Without argument, a default prefix is used to build the path. An optional string prefix can control the original name.
- *absolute-path → String (String+)*
  The *absolute-path* function returns an absolute path name from an argument list. Without argument, the command returns the root directory name. With one or several argument, the absolute path is computed from the root directory.
- *relative-path → String (String+)*
  The *relative-path* function returns a relative path name from an argument list. With one argument, the function returns it. With two or more arguments, the relative path is computed by joining each argument with the previous one.
- *rmfile → none (String+)*
  The *rmfile* function removes one or several files specified as the arguments. If one file fails to be removed, an exception is raised.
- *mkdir → none (String+)*
  The *mkdir* function creates one or several directories specified as the arguments. If one directory fails to be created, an exception is raised.
- *mhdir → none (String+)*
  The *mhdir* function creates hierarchically one or several directories specified as the arguments. If one directory fails to be created, an exception is raised.
- *rmdir → none (String+)*
  The *rmdir* function removes one or several directories specified as the arguments. If one directory fails to be removed, an exception is raised.
- *get-base-name → String (String)*
  The *get-base-name* function returns the base name from a path. The base name can be either a file name or a directory name. By definition, a path is made of a base path and a base name.
- *get-base-path → String (String)*
  The *get-base-path* function returns the base path from a path. The base path is a directory name. By definition, a path is made of a base path and a base name.
- *get-extension → String (String)*
  The *get-extension* function returns the extension from a path.
- *remove-extension → String (String)*
  The *remove-extension* function returns the extension from a path. In order to get a base file name from a path, the *get-base-name* function must be called first.

20. Object Directory

The *Directory* class provides some facilities to access a directory. By default, a directory object is constructed to represent the current directory. With one argument, the object is constructed from the directory name. Once the object is constructed, it is possible to retrieve its content.

20.1. Predicate.

- directory-p

20.2. Inheritance.

- Object

20.3. Constructors.

- *Directory → (none)*
  The *Directory* constructor creates a directory object those location is the current directory. If the directory cannot be opened, an exception is raised.
- *Directory → (String)*
  The *Directory* constructor create a directory object by name. If the directory cannot be opened, an exception is raised. The first argument is the directory name to open.

20.4. Methods.

- *mkdir → Directory (String)*
  The *mkdir* method creates a new directory in the current one. The full path is constructed by taking the directory name and adding the argument. Once the directory is created, the method returns a directory object of the newly constructed directory. An exception is thrown if the directory cannot be created.
- *rmdir → none (String)*
  The *rmdir* method removes an empty directory. The full path is constructed by taking the directory name and adding the argument. An exception is thrown if the directory cannot be removed.
- *rmfile → none (String)*
  The *rmfile* method removes a file in the current directory. The full path is constructed by taking the directory name and adding the argument. An exception is thrown if the file cannot be removed.
- *get-name → String (none)*
  The *get-name* method returns the directory name. If the default directory was created, the method returns the full directory path.
- *get-list → List (none)*
  The *get-list* method returns the directory contents. The method returns a list of strings. The list contains all valid names at the time of the call, including the current directory and the parent directory.
- *get-files → List (none)*
  The *get-files* method returns the directory contents. The method returns a list of strings of files. The list contains all valid names at the time of the call.
- *get-subdirs → List (none)*
  The *get-subdirs* method returns the sub directories. The method returns a list of strings of sub-directories. The list contains all valid names at the time of the call, including the current directory and the parent directory.
- *next-name → String (none)*
  The *next-name* method returns the next available name from the directory stream. This method is useful when operating with a large number of elements.

- *next-path → String (none)*
  The *next-path* method returns the next available path name from the directory stream. This method is useful when operating with a large number of elements.
- *next-file-name → String (none)*
  The *next-file-name* method returns the next available file name from the directory stream. This method is useful when operating with a large number of elements.
- *next-file-path → String (none)*
  The *next-file-path* method returns the next available file path name from the directory stream. This method is useful when operating with a large number of elements.
- *next-dir-name → String (none)*
  The *next-dir-name* method returns the next available directory name from the directory stream. This method is useful when operating with a large number of elements.
- *next-dir-path → String (none)*
  The *next-dir-path* method returns the next available directory path name from the directory stream. This method is useful when operating with a large number of elements.

21. Object Logtee

The *Logtee* class is a message logger facility associated with an output stream. When a message is added to the logger object, the message is also sent to the output stream, depending on the controlling flags. The name "logtee" comes from the contraction of "logger" and "tee". One particularity of the class is that without a stream, the class behaves like a regular logger.

21.1. Predicate.

- logtee-p

21.2. Inheritance.

- Logger

21.3. Constructors.

- *Logtee → (none)*
  The *Logtee* constructor creates a default logger without an output stream
- *Logtee → (Integer)*
  The *Logtee* constructor creates a logger object with a specific size without an output stream.
- *Logtee → (Output)*
  The *Logtee* constructor creates a logger object with an output stream.
- *Logtee → (Integer Output)*
  The *Logtee* constructor creates a logger object with a specific size and an output stream. The first argument is the logger window size. The second argument is the output stream.
- *Logtee → (Integer String Output)*
  The *Logtee* constructor creates a logger object with a specific size, an information string and an output stream. The first argument is the logger window size. The second argument is the logger information string. The third argument is the output stream.

21.4. Methods.

- *set-output-stream → none (Output)*
  The *set-output-stream* method attaches the output stream to the logtee object.
- *get-output-stream → Output (none)*
  The *get-output-stream* method returns the logtee output stream.
- *set-tee → none (Boolean)*
  The *set-tee* method sets the logtee control flag. The control flag controls the message display to the output stream.
- *get-tee → Boolean (none)*
  The *get-tee* method returns the logtee output stream.

22. Object NamedFifo

The *NameFifo* class is a string vector designed to operate as a stream fifo object. The class provides the facility to read or write the fifo content from a stream. The stream can be created by name for writing, in which case the named fifo operates as a backup object.

22.1. Predicate.

- named-fifo-p

22.2. Inheritance.

- StrvecNameable

22.3. Constructors.

- *NamedFifo → (none)*
  The *NamedFifo* constructor creates a default named fifo without a backing name. In this case the fifo cannot be read or written by stream.
- *NamedFifo → (String)*
  The *NamedFifo* constructor creates a named fifo by name. The name is used as a file name for reading or writing the fifo.
- *NamedFifo → (String Boolean)*
  The *NamedFifo* constructor creates a named fifo by name. The name is used as a file name for reading or writing the fifo.If the boolean argument is true, the fifo is read.

22.4. Methods.

- *read → none (none)*
  The *read* method reads the fifo file name and fill the fifo.
- *write → none (none)*
  The *write* method writes the fifo contents to the fifo file name.
- *set-name → none (String)*
  The *set-name* method sets the fifo file name.

23. Object FileInfo

The *FileInfo* is a file information class that holds the primary information related to a file, such like its size or its modification time. The file information is set at construction but can be updated with the help of the *update* method.

23.1. Predicate.

- file-info-p

23.2. Inheritance.

- Nameable

23.3. Constructors.

- *→ (String)*
  The *FileInfo* constructor creates a file information by name. The string argument is the file name to query.

23.4. Methods.

- *length → Integer (none)*
  The *length* method returns the file size information.
- *get-modification-time → Integer (none)*
  The *get-modification-time* method returns the file modification time. The time can be used as an argument to the *Time* or *Date* object.
- *update → none (none)*
  The *update* method the file information data.

CHAPTER 15

Standard Spreadsheet Module

The *Standard Spreadsheet* module is an original implementation that provides the necessary objects for designing a spreadsheet. A spreasheet acts a great interface which structure data in the form of record and sheets. Once structured, these data can be indexed, manipulated and exported into various formats.

1. Spreadsheet concepts

The sole purpose of using a spreadsheet is to collect various data and store them in such a way that they can be accessed later. Unlike standard spreadsheet system, the standard spreadsheet module does not place restrictions on the data organization. The spreadsheet module stores data in a hierarchical fashion. The basic data element is called a *cell* . A set of cells is a *record* . A set of records is a *sheet* . A set of sheets and records is a *folio* .

1.1. Cell and data. A *cell* is a data container. There is only one data element per cell. Eventually a name can be associated with a cell. The cell data can be any kind of literals. Such literals are integer, real, boolean, character or strings.

1.2. Record. A *record* is a vector of cells. A record can be created by adding cell or simply by adding data. If the record has a predefined size, the cell or data can be set by indexing.

1.3. Sheet. A *sheet* is a vector of records. A sheet can be created by adding record. Similarly, if the sheet has a predefined size, record cell or data can be added by indexing. A sheet can also be seen as a 2 dimensional array of cells. For the purpose of managing extra information, the sheet carry also several extra records, namely, the *marker record* , the *header record* and *footer record* as well as the *tag vector* and the *signature* .

1.4. Folio. A *folio* is a set of sheets and/or records. A folio of sheets permits to structure data in the form of tables. Since cell, record and table can have a name, it is possible to create link between various elements, thus creating a collection of structured data.

2. Storage model

There are several ways to integrate data. In the simplest form, data are integrated in a record list. A complex model can be built with a sheet. More complex models can also be designed by using a folio.

2.1. Single record model. With a single record model, the data are accumulated in a single array. This kind of data storing is particularly adapted for single list recording. As a single record, the basic search and sorting algorithm can be applied. For instance, a list name can be stored as a single record. With this view, there is no difference between a list, a vector and a record. The record can also be named.

2.2. Record importation. Data are imported into the record, either by construction, list or stream. Since the record object is a serializeable object, the importation process is also performed automatically in the collection. The base record importation class implements a simple importation model based on blank separated literals. Complex importation models can be devised by derivation. A special case with a cons cell is also supported where the *car* is the cell name and the *cadr* is the cell object.

```
# an example of file importation
1 "a string" 'a'
'b' ("cell name" 2) 123
```

The previous example shows the file structure that can be used to import cell data. The fist line defines a record with 3 cells. The second line defines also a record with 3 cells. The second cell is a named cell.

2.3. Record exportation. A record is an object that can be serialized. It can therefore be exported easily. However, in the serialized form, the record is in a binary form. It is also possible to walk through the record and exports, for each cell its literal form.

3. Folio indexation

There are various ways to access a folio by reference. Since a folio can contain several sheets, it seems natural to access them by tag. The other method is to index the cells in a cross-reference album in order to access rapidly.

3.1. Sheet access model. The sheet access model uses a tag to access one or several sheets in a folio. A tag is a string attached to a sheet. It is possible in a folio to have several sheet with the same tag. It is also possible to attach several tags to a sheet. When a folio is searched by tag, the first sheet that matches the tag is said to be the valid one. If all sheets that match the requested tag are needed, it is possible to create a derived folio with all sheets that match the requested tag.

3.2. Cell access model. The cell access model operates with a cross-reference table built with an index. An index is a multiple entry record that stores the cell location. A cell coordinate comprises the cell index in the record, the record index in the sheet and the sheet index in the folio. If an index contains multiple entries, this indicates that several cells are indexed. A cell cross-reference table is a collection of index. Generally the index name is the cell name. When the cross-reference table is built, all cell of interests are scanned and if a cell name exists, the cell is indexed in the cross-reference table. If there are several cells with the same name, the index length associated with the name is the number of cells with that name.

3.3. Search and access. The methodology for searching is to decide whether a sheet or a cell should be accessible. If a sheet access is desired, the search by tag method is the preferred way. This method assumes that the requested sheet is structured in a particular way, known to the user. If a cell access seems more appropriate, a cross-reference table should be built first, and the search done from it. In the case of search by tag, the method is dynamic and operates well when sheets are added in a folio. When a cross-reference table is used, proper care should be taken to rebuild the cross-reference table when some sheets are added unless the user knows that there is no need for it.

4. Folio object

The *Folio* object is the primary object used for storing data. Although, a folio is a collection of sheets, it the primary object that should be created when manipulating such collection.

4.1. Creating a folio. The *Folio* object is built without argument. In this case, the folio is empty. A predicate is available for testing the *Folio* object.

```
const sps (afnix:sps:Folio)
afnix:sps:folio-p sps # true
```

The constructor can operate also by name or by input stream. With a string, a new folio those name is the argument is created. By stream, a new folio is created and loaded with the input stream data. Eventually, the folio name can be set with the *set-name* command and retrieved with the *get-name* command.

```
const sps (afnix:sps:Folio)
sps:set-name "planets"
```

5. Sheet object

The *Sheet* object is the primary object used to store data in a folio. Since a *Folio* object is a collection of sheets, a sheet can be manipulated either by getting getting it from the folio or by creating it independently and adding it into the folio.

5.1. Creating a sheet. An empty sheet can be created simply with or without name. Without argument, an unnamed sheet is created. Similar to the *Folio* class, the sheet name can be passed at construction or set with the *set-name* method. As usual a predicate is provided.

```
const sht (afnix:sps:Sheet)
afnix:sps:sheet-p sht # true
```

When the sheet is created, it can be added to the folio spreadsheet with the *add* method.

```
const sht (afnix:sps:Sheet "data")
sps:add sht
```

5.2. Adding data to the sheet. The process of adding data to a sheet is a straightforward operation with the *add-data* method or the *add* method. With the *add-data* method, data are added as literals. With the *add* method, data are added with the help of a record object.

```
sht:add-data "Mercury" 4840 "1407:36"
sht:add-data "Venus" 12400 "5819:51"
sht:add-data "Earth" 12756 "23:56"
sht:add-data "Mars" 6800 "24:37"
sht:add-data "Jupiter" 142800 "9:50"
sht:add-data "Saturn" 120800 "10:14"
sht:add-data "Uranus" 47600 "10:49"
sht:add-data "Neptune" 44600 "15:40"
sht:add-data "Pluto" 5850 "153:17"
sht:add-data "Sedna" 1800 "960:00"
```

Data can be imported in a sheet by importation with an input stream. During the importation, the serialized data are decoded and placed sequentially in the stream.

5.3. Sheet sorting. A sheet can be sorted with the *sort* method. The *sort* method uses the first integer argument as the column number. The second optional argument is a boolean argument that selects the sorting method which can be ascending (by default) or descending if the flag is false.

```
sht:sort 0
sht:sort 1 false
```

6. Record object

The *Record* object is an alternative to the sheet data filling. With the help of the *add* method, a record can be added to a sheet.

6.1. Creating a record. A record is a named object that acts as a vector of cells. The record name can be set either by construction or with the *set-name* method. As usual a predicate is provided.

```
const rcd (afnix:sps:Record)
afnix:sps:record-p # true
```

6.2. Filling a record. A record can be filled either with literal data or *Cell* objects. In the first case, the cell is automatically created for each literal added to the record. The *add* method add a cell or literal to the record.

```
const rcd (Record)
rcd:add "Mercury" 4840 "1407:36"
```

For data extraction, the *length* method returns the record length. Data can be extracted by index with either the *get* or *map* method. The *get* method returns the record cell while the *map* method returns the cell literal.

7. Object search

The search methodology is divided either by sheet or cells. In a folio, the search is done in terms of sheets while the construction of a cross-reference table is required for searching cells.

7.1. Searching in a folio. The *Folio* object provides the primary mean to search for sheet. The *find* and *lookup* methods return a sheet by tag. The first sheet that matches the tag is returned by these methods. The *find* method returns nil if the sheet cannot be found while the *lookup* method throws an exception.

```
# get a sheet by tag
const sheet (folio:lookup "the tag")
```

If there are several sheets that matched the tag, the *filter* method is more appropriate. The *filter* method returns a new folio that contains the sheet that matches the requested tag.

```
# get a sub-folio by tag
const sf (folio:filter "the tag")
```

7.2. Searching for a cell. The *Folio* object also provides the method for building a cross-reference table. The cross-reference table is represented by the *Xref* object. By default, a complete *Xref* object table is built with the *getxref* folio method. Such method, scan all cells in the folio and add them in the cross-reference table if the cell has a name.

```
# get a default xref table
const xref (folio:getxref)
```

The cross-reference table can also be built by searching for sheet tags. With a string argument, the *getxref* search for all sheets that matches the tag and then build a cross-reference table. This method is equivalent to the operation of building a new folio by tag with the *filter* method and then building the cross-reference table.

```
# get a xref table by tag
const xref (folio:getxref "the tag")
# another method
const sf (folio:filter "the tag")
const xref (sf:getxref)
```

A cross-reference table can also be built by cell index and/or record index. With a cell index, the cross-reference table is built by indexing the sheet column. With both the cell and record indexes, the cross-reference table is built by indexing all cells at the coordinate for all sheets.

```
# get a xref table by cell index
const xref (folio:getxref 0)
# get a xref table by cell and record index
const xref (folio:getxref 0 1)
```

CHAPTER 16

Standard Spreadsheet Reference

1. Object Cell

The *Cell* class is a data container. There is only one data element per cell. Eventually a name can be associated with a cell. The cell data can be any kind of literals. Such literals are integer, real, boolean, character or strings. A cell is the basic block used to build a spreadsheet.

1.1. Predicate.

- cell-p

1.2. Inheritance.

- Nameable

1.3. Constructors.

- *Cell → (none)*
  The *Cell* constructor create a default cell with no name and no value. When evaluated, the cell returns nil.
- *Cell → (Literal)*
  The *Cell* constructor create a default cell by value. The argument is a literal object which can be viewed with its string representation.
- *Cell → (String Literal)*
  The *Cell* constructor create a default cell by name and value. The first argument is the cell name. The second argument is a literal object which can be viewed with its string representation.

1.4. Methods.

- *get → Literal (none)*
  The *get* method returns the cell literal value.
- *set → none (Literal)*
  The *set* method sets the cell literal value.
- *get-name → String (none)*
  The *get-name* method returns the cell name.
- *set-name → none (String)*
  The *set-name* method sets the cell name.

2. Object Persist

The *Persist* class is a base class for the **AFNIX** spreadsheet module. The class defines the methods that are used to read of write a particular persistent object. When saving, the object name is retreived with the get name method. The saveas method does the same as save but takes a file name argument.

2.1. Predicate.

- persist-p

2.2. Inheritance.

- Nameable

2.3. Methods.

- *save → none (none)*
  The *save* method saves the current object.
- *saveas → none (String)*
  The *saveas* method saves the current object into the file those name is the string argument.

3. Object Record

The *Record* class is a cell container. A record can be created by adding cell or simply by adding data. If the record has a predefined size, the cell or data can be set by indexing. A name can be attached to the record. Facilities are provided to access directly the cell associated with the record. A record can also be created by name.

3.1. Predicate.

- record-p

3.2. Inheritance.

- Persist

3.3. Constructors.

- *Record → (none)*
  The *Record* constructor create an empty record with no name and no cell.
- *Record → (String)*
  The *Record* constructor create an empty record by name name. The argument is the record name to use.

3.4. Methods.

- *add → none (Cell—Literal...)*
  The *add* method adds one or several cell objects to the record. In the first form, the argument is a cell that is added in the record. In the second form, the argument is a literal that is used to create a cell by value.
- *get → Cell (Integer)*
  The *get* method returns a record cell by index. The index must be within the record range or an exception is raised.
- *map → Literal (Integer)*
  The *map* method map a record cell by index. The index must be within the record range or an exception is raised. The cell is mapped to a literal object.
- *set → none (Integer Cell—Literal)*
  The *set* method sets the record cell by index. The record index must be valid in order to succeed. A new cell is created prior the record update. The previous cell data is lost including its name.
- *sort → none (none—Boolean)*
  The *sort* method sorts a record in place. Without argument, the record is sorted in ascending order. The optional boolean argument specifies the sorting mode. If true, the record is sorting in ascending order and in descending order if false.
- *find → Cell (String)*
  The *find* method finds a cell by name. If the cell is not found, the nil object is returned. This method is similar to the *lookup* method.
- *get-index → Integer (String)*
  The *get-index* method finds a cell index by name. If the cell is not found, an exception is raised. The index is the cell position in the record.
- *lookup → Cell (String)*
  The *lookup* method finds a cell by name. If the cell is not found, an exception is raised. This method is similar to the *find* method.
- *length → Integer (none)*
  The *length* method returns the record length.
- *get-name → String (none)*
  The *get-name* method returns the record name.

- *reset → none (none)*
  The *reset* method rest the record. The record name is not changed but all record cells are removed.
- *set-name → none (String)*
  The *set-name* method sets the record name.

4. Object Sheet

The *Sheet* class is a record container. A sheet can be created by adding records. Similarly, if the sheet has a predefined size, record cell or data can be added by indexing. A sheet can be also seen as a 2 dimensional array of cells. Like a record, a sheet can be named. Without argument, a default sheet is created. With a string argument, the sheet is created with an initial name.

4.1. Predicate.

- sheet-p

4.2. Inheritance.

- Persist

4.3. Constructors.

- *Sheet → (none)*
  The *Sheet* constructor create an empty sheet with no name and no cell.
- *Sheet → (String)*
  The *Sheet* constructor create an empty sheet by name. The argument is the sheet name to use.
- *Sheet → (String String)*
  The *Sheet* constructor create an empty sheet by name and info. The first argument is the sheet name to use. The second argument is the sheet information string.

4.4. Methods.

- *add → none (Record)*
  The *add* method adds a record to the sheet.
- *get → Record (Integer)*
  The *get* method returns a sheet record by index. The index must be within the sheet range or an exception is raised.
- *set → none (Integer Record)*
  The *set* method sets the sheet record by index. The sheet index must be valid in order to succeed.
- *sort → none (none—Integer—Boolean—Integer Boolean)*
  The *sort* method sorts the sheet in place. By default, the sheet is sorted in ascending order with column 0. With one argument, the argument can be either the sorting mode or the column number. If the mode is true, the sheet is sorted in ascending mode. If the mode is false, the sheet is sorted in descending mode. With two arguments, the first argument is the column number and the second is the sorting mode.
- *length → Integer (none)*
  The *length* method returns the sheet length.
- *convert → PrintTable ([Integer [Integer [Boolean]]])*
  The *convert* method converts the sheet into a print-table representation. Without argument, the whole sheet is converted. With one argument, the sheet is converted unto a maximum of rows. The second optional argument is the start index that default to 0. The third optional argument is the literal format. By default, the literal is formatted as a string. If the flag is true, the literal is formatted as a literal string representation.
- *add-data → none ([Cell—Literal]+)*
  The *add-data* method adds one or several cell object to a sheet record. The sheet record is initially created and the object elements are added sequentially to the

record. In the first form, the argument is a cell that is added in the record. In the second form, the argument is a literal that is used to create a cell by value.

- *add-marker → none ([Literal]+)*
  The *add-marker* method adds one or several literal objects to a sheet marker record.
- *marker-length → Integer (none)*
  The *marker-length* method returns the number of markers defined for this sheet.
- *get-marker → Cell (Integer)*
  The *get-marker* method returns a marker cell by index from the sheet marker record.
- *set-marker → none (Integer Literal)*
  The *set-marker* method set the sheet marker record by index and literal.
- *find-marker → Integer (String)*
  The *find-marker* method find the marker index. The argument is the marker string value.
- *add-sign → none ([Literal]+)*
  The *add-sign* method adds one or several literal objects to a sheet sign record.
- *signature-length → Integer (none)*
  The *signature-length* method returns the number of signs defined for this sheet.
- *get-sign → Cell (Integer)*
  The *get-sign* method returns a sign cell by index from the sheet sign record.
- *set-sign → none (Integer Literal)*
  The *set-sign* method set the sheet sign record by index and literal.
- *find-sign → Integer (String)*
  The *find-sign* method find the sign index. The argument is the sign string value.
- *add-header → none (Cell—Literal...)*
  The *add-header* method adds one or several cell object to a sheet header record. In the first form, the argument is a cell that is added in the record. In the second form, the argument is a literal that is used to create a cell by value.
- *get-header → Cell (Integer)*
  The *get-header* method returns a header cell by index from the sheet header record.

- *map-header → Literal (Integer)*
  The *map-header* method maps to a literal object a header cell by index from the sheet header record.
- *set-header → none (Integer Literal)*
  The *set-header* method set the header record by index and literal. The cell associated with the header record is updated with the literal value.
- *add-footer → none ([Cell—Literal]+)*
  The *add-footer* method adds one or several cell object to a sheet footer record. In the first form, the argument is a cell that is added in the record. In the second form, the argument is a literal that is used to create a cell by value.
- *get-footer → Cell (Integer)*
  The *get-footer* method returns a footer cell by index from the sheet footer record.

- *map-footer → Literal (Integer)*
  The *map-footer* method maps to a literal object an footer cell by index from the sheet footer record.
- *set-footer → none (Integer Literal)*
  The *set-footer* method set the footer record by index and literal. The cell associated with the footer record is updated with the literal value.

- *get-name → String (none)*
  The *get-name* method returns the sheet name.
- *set-name → none (String)*
  The *set-name* method sets the sheet name.
- *get-info → String (none)*
  The *get-info* method returns the sheet info.
- *set-info → none (String)*
  The *set-info* method sets the sheet info.
- *add-tag → none ([String]+)*
  The *add-tag* method adds a tag into the tags vector. The tag can be used to mark a sheet in a folio.
- *tag-p → Boolean (string)*
  The *tag-p* method returns true if the given tag is defined for this sheet.
- *tag-length → Integer (none)*
  The *tag-length* method returns the number of tags defined for this sheet.
- *get-tag → String (Integer)*
  The *get-tag* method returns a tag by index.
- *set-tag → none (Integer Literal)*
  The *set-tag* method set the sheet tag record by index and literal.
- *find-tag → Integer (String)*
  The *find-tag* method find the tag index. The argument is the tag string value.
- *reset → none (none)*
  The *reset* method resets the sheet. The name and information is unchanged during this operation.

5. Object Folio

The *Folio* class is a sheet container. A folio of sheets can be structured to handle various data organization. Since all objects are named, it is possible to index them for fast data search. An empty folio can be created by name or filled with an input stream.

5.1. Predicate.

- folio-p

5.2. Inheritance.

- Persist

5.3. Constructors.

- *Folio → (none)*
  The *Folio* constructor create an empty folio with no name and no cell.
- *Folio → (String)*
  The *Folio* constructor create an empty folio by name. The argument is the folio name to use.
- *Folio → (String String)*
  The *Folio* constructor create an empty folio by name and info. The first argument is the folio name to use. The second argument is the folio information string.
- *Folio → (InputStream)*
  The *Folio* constructor create an empty folio and fill it by reading the data from the input stream. The input stream must have the data in a serialized form.

5.4. Methods.

- *write → none (OutputStream)*
  The *write* method write the folio contents to an output stream. The written form is a serialized form.
- *add → none (Sheet)*
  The *add* method adds a sheet to the folio.
- *get → Sheet (Integer)*
  The *get* method returns a sheet by index. The index must be within the folio range or an exception is raised.
- *set → none (Integer Sheet)*
  The *set* method sets the folio by index and sheet. The index must be within the folio range or an exception is raised.
- *tag-p → Boolean (String)*
  The *tag-p* method returns true if a sheet with the tag argument exists in the folio.
- *find → Sheet (String)*
  The *find* method finds a sheet by tag. The first found sheet those tag is matched is returned. In the case that multiple sheet share the same tag, the *filter* should be used. If no sheet is found the method return the nil object.
- *lookup → Sheet (String)*
  The *lookup* method finds a sheet by tag. The first found sheet those tag is matched is returned. In the case that multiple sheet share the same tag, the *filter* should be used. If no sheet is found the method throws an exception. This method is similar to the *find* method.
- *filter → Folio (String)*
  The *filter* method return a new folio with sheets that match the argument tag. If no sheet is found, the folio is empty.

- *reset → none (none)*
  The *reset* method resets the folio. The name and information is unchanged during this operation.
- *length → Integer (none)*
  The *length* method returns the folio length.
- *get-name → String (none)*
  The *get-name* method returns the folio name.
- *set-name → none (String)*
  The *set-name* method sets the folio name.
- *get-info → String (none)*
  The *get-info* method returns the folio info.
- *set-info → none (String)*
  The *set-info* method sets the folio info.
- *get-xref → Xref (none—Integer—String—Integer Integer)*
  The *get-xref* method returns a cross-reference table from the folio. Without argument, the whole folio is scanned and all named cells are added in the cross-reference table. With an integer argument, all cells that matches the cell index argument are added in the cross-reference table. With a string argument, all cells that have the same name are added in the table. Finally, with two arguments that represents the cell index and the record index are used, all cells are added in the table with these two coordinates.

6. Object Index

The *Index* class is a class that maintain a cell index at the folio level. A cell index is composed of the sheet index, the record index and the cell index. The index object can be used to access in a generic way a particular cell. Additionally, the folio name can also be stored in the index. It is possible to have multiple records that represents the same cell.

6.1. Predicate.

- index-p

6.2. Inheritance.

- Object

6.3. Constructors.

- *Index → (none)*
  The *Index* constructor creates an empty index.
- *Index → (Integer)*
  The *Index* constructor creates an index with a cell index as its coordinate.
- *Index → (Integer Integer)*
  The *Integer* constructor creates an index with a cell and record indexes as its coordinate. The first argument is the cell index. The second argument is the record index.
- *Index → (Integer Integer Integer)*
  The *Index* constructor creates an index with a cell, record and sheet indexes as its coordinate. The first argument is the cell index. The second argument is the record index. The third argument is the sheet index.

6.4. Methods.

- *add → none (Integer—Integer Integer—Integer Integer Integer)*
  The *add* method adds a new index coordinate in the index object. In the first form, the cell index is used as the coordinate. In the second form, the cell and record indexes are used as the coordinate. In the third form, the cell, record and sheet indexes are used as the coordinate.
- *reset → none (none)*
  The *reset* method reset the index by removing all attached coordinates.
- *length → Integer (none)*
  The *length* method returns the index length. The index length is the number of coordinate entries in the index.
- *exists-p → Boolean (Integer—Integer Integer—Integer Integer Integer)*
  The *exists-p* predicate returns true if a coordinate entry exists in the index. In the first form, the cell index is used as the coordinate. In the second form, the cell and record indexes are used as the coordinate. In the third form, the cell, record and sheet indexes are used as the coordinate.
- *set-index-cell → none (Integer Integer)*
  The *set-index-cell* method sets the cell index by position. The first argument is the coordinate position. The second argument is the cell index to use.
- *update-index-cell → none (Integer)*
  The *update-index-cell* method updates the cell index for all entries in the index. The argument is the new cell index to use for the update process.
- *get-index-cell → Integer (Integer)*
  The *get-index-cell* method returns the cell index for a particular entry. The argument is the entry position.

- *set-index-record → none (Integer Integer)*
  The *set-index-record* method sets the record index by position. The first argument is the coordinate position. The second argument is the record index to use.
- *update-index-record → none (Integer)*
  The *update-index-record* method updates the record index for all entries in the index. The argument is the new record index to use for the update process.
- *get-index-record → Integer (Integer)*
  The *get-index-record* method returns the record index for a particular entry. The argument is the entry position.
- *set-index-sheet → none (Integer Integer)*
  The *set-index-sheet* method sets the sheet index by position. The first argument is the coordinate position. The second argument is the cell sheet to use.
- *update-index-sheet → none (Integer)*
  The *update-index-sheet* method updates the sheet index for all entries in the index. The argument is the new sheet index to use for the update process.
- *get-index-sheet → Integer (Integer)*
  The *get-index-sheet* method returns the sheet index for a particular entry. The argument is the entry position.

7. Object Xref

The *Xref* class is a cross-reference class. The class maintains the association between a name and an index. with a particular name, an index entry is created if it does not exists. Such entry can be later used to access the cell content by index.

7.1. Predicate.

- xref-p

7.2. Inheritance.

- Object

7.3. Constructors.

- *Xref → (none)*
  The *Xref* constructor creates an empty cross-reference object.

7.4. Methods.

- *add → none (String [Integer—Integer Integer—Integer Integer Integer])*
  The *add* method adds a new reference in the table. The first argument is always the index name. In the first form, the cell index is used as the coordinate. In the second form, the cell and record indexes are used as the coordinate. In the third form, the cell, record and sheet indexes are used as the coordinate.
- *get → Index (Integer—String)*
  The *get* method returns an *Index* object either by name or position. With an integer argument, the index is obtained by position. With a string argument, the index with the matching name is returned.
- *reset → none (none)*
  The *reset* method resets the cross-reference table.
- *length → Integer (none)*
  The *length* method returns the length of the cross-reference table.
- *lookup → Index (String)*
  The *lookup* method returns an index those name is the matching argument. If the index cannot be found, an exception is raised.
- *exists-p → Boolean (String)*
  The *exists-p* predicate returns true if an index those name is the matching argument exists in the cross-reference table.
- *get-name → String (Integer)*
  The *get-name* method returns the index name by position.

CHAPTER 17

Standard System Access Module

The *Standard System Access* module is an original implementation of various objects designed to provide a specialized access to the underlying system. Most of the system accesses are provided in the form of functions which have been designed to be portable as possible. One example of this, are the time and date management objects.

1. Interpreter information

The interpreter provides a set reserved names that are related to the system platform. Example `0501.als` demonstrates the available information.

```
zsh> axi 0501.als
program name : afnix
operating system name : linux
operating system type : unix
afnix official uri : http://www.afnix.org
```

1.1. Interpreter version. The interpreter version is identified by 3 numbers called *major* , *minor* and *patch* numbers. A change in the major number represents a major change in the writing system. The minor number indicates a major change in the interface or libraries. A change in the patch number indicates bug fixes. All values are accessed via the interpreter itself. The *major-version* , *minor-version* , *patch-version* symbols are bound to these values.

```
println "major version number : "
interp:major-version
println "minor version number : "
interp:minor-version
println "patch version number : "
interp:patch-version
```

1.2. Operating system. The operating system is uniquely identified by its name. The operating system type (or category) uniquely identifies the operating system flavor.

```
println "operating system name : "
interp:os-name
println "operating system type : "
interp:os-type
```

1.3. Program information. Program information are carried by two symbols that identifies the program name and the official uri. While the first might be useful, the second one is mostly used by demo programs.

```
println "program name : "
interp:program-name
println "afnix official uri : "
interp:afnix-uri
```

Function	Description
exit	terminate with an exit code
sleep	pause for a certain time
get-pid	get the process identifier
get-env	get an environment variable
get-host-name	get the host name
get-user-name	get the user name

2. System services

The *system services* module provides various functions that cannot be classified into any particular category.

The *exit* function terminates the program with an exit code specified as the argument. The *sleep* function pause the specific thread for a certain time. The time argument is expressed in milliseconds. The *get-pid* function returns the process identifier. The *get-env* function returns the environment variable associated with the string argument. The *get-host-name* function returns the host name. The host name can be either a simple name or a canonical name with its domain, depending on the system configuration. The *get-user-name* function returns the current user name.

3. Time and date

The *Time* and *Date* classes are classes designed to manipulate time and date. The writing system operates with a special coordinated time which uses the reference of Jan 1st 0000 in a modified proleptic Gregorian calendar. This proleptic feature means that the actual calendar (Gregorian) is extended beyond year 1582 (its introduction year) and modified in order to support the year 0. This kind of calendar is somehow similar to the astronomical Gregorian calendar except that the reference date is 0 for the writing system. This method presents the advantage to support negative time. It should be noted that the 0 reference does not means year 1BC since year 0 did not exist at that time (the concept of zero is fairly new) and more important, the date expressed in the form 1BC generally refers to the Julian calendar since the date is before 1582. Although, the class provides several methods to access the time and date fields, it is also possible to get a string representation that conforms to ISO-8601 or to RFC-2822.

3.1. Time and date construction. By default, a time instance of current time is constructed. This time reference is obtained form the machine time and adjusted for the internal representation. One feature of this class is that the time instance does not have to be bounded with 24 hours. The time stored is the absolute time, which should be considered like a temporal reference – or date – those origin is 0 in some calendar representation.

```
1 const time (afnix:sys:Time)
2 assert true (afnxi:sys:time-p time)
```

A simple time representation can also be built by hours, minutes and seconds. In this case, the time is a time definition at day 0 in the reference calendar.

```
1 const time (afnix:sys:Time 12 23 54)
```

By default a date instance of the current date is constructed. The current date is computed from the machine time and expressed in a particular calendar. By default, the engine uses a special Gregorian calendar as explained before. The important point here s that the date will show up like the user should expect.

```
1 const date (afnix:sys:Date)
2 assert true (afnix:sys:date-p date)
```

A date instance can also be built with an absolute time expressed in seconds or with specific elements. with one argument, the date is expressed in seconds since the origin. Since the internal representation is 64 bits, the date room is quite large. For example, the absolute time to represent Jan 1st 1970 is 62167219200 seconds. This *epoch* is used to adjust the system time on some UNIX system. Another way to create a specific date is to use the date descriptor by year, month and day. With 6 arguments, the time components can also be given. This makes *Date* one of the constructor that accept the largest number of arguments.

```
1 const date (afnix:sys:Date 1789 7 14 16 0 0)
2 assert true (afnix:sys:date-p date)
```

In the previous example, at 17:00 local time, 16:00Z although the concept of time zone was not formalized, the Bastille surrenders on July 14 1789. This example shows that extreme care should be used when dealing with old dates. Note that a simpler form could have been used to set that date. With 3 argument, the date is set at time 00:00:00Z.

```
1 const date (afnix:sys:Date 1789 7 14)
2 assert true (afnix:sys:date-p date)
```

3.2. Time and date representation. Except for some special applications – like the cookie maximum age –, the date representation is quite standard and can be found either in the form of ISO-8601 or RFC-2822.

```
1 const time (afnix:sys:Time 12 44 55)
2 println (time:format) # 12:44:55
3 println (time:to-iso) # 14:44:55
4 println (time:to-rfc) # 14:44:55 +0200
```

in the first form, the time is represented naturally by hour, minutes and seconds. By default, it is the local time that is given. With a flag set to true, the UTC time is displayed. In the second form, the time is displayed in the ISO-8601 form which is the same as before. In the third form, the time is displayed in the RFC-2822 form. This form is always expressed locally with the timezone difference associated with it. It shall be noted that the ISO-8601 mandate to use the suffix 'Z' for the zulu time. This is the difference when using the *true* flag with the *format* and *to-iso* methods.

```
1 println (time:format true) # 12:44:55
2 println (time:to-iso true) # 12:44:55Z
```

The date representation also operates with 3 methods, namely *format* , *to-iso* and *to-rfc* . For example, if the time is 12:00 in Paris on July 14th 2000, the date will be displayed like below.

```
1 const date (afnix:sys:Date 2000 7 14 12 0 0)
2 # Fri Jul 14 07:00:00 2000
3 println (date:format)
4 # 2000-07-14T07:00:00
5 println (date:to-iso)
6 # Fri, 14 Jul 2000 07:00:00 -0500
7 println (date:to-rfc)
```

The example show the local time. With UTC display, only the first two methods can be used.

```
1 const date (afnix:sys:Date 2000 7 14 12 0 0)
2 println (date:format true) # Fri Jul 14 12:00:00 2000
3 println (date:to-iso true) # 2000-07-14T12:00:00Z
```

4. Options parsing

The *Options* class provides a convenient mechanism to define a set of options and to parse them in a simple way. The object is constructed by specifying which option is valid and how it behaves. The arguments can be passed to the object for subsequent analysis. An option can be either a unique option or a string option. In this later case, multiple value for the same option can be accepted. In that case, the option is said to be a string vector option. An option can be also an option list. I that case, the option is defined with a set of valid string. A list option is associated with a boolean flag for each string defined with that option.

4.1. Option creation. An *Options* is created by invoking the constructor with or without a user message. The user message is used by the *usage* method which display an information message.

```
const options (
  afnix:sys:Options "axi [options] [file [arguments]]")
```

Eventually, the *set-user-message* method can be used to set the user message.

4.2. Options definition. The process of defining options is done by specifying the option character, eventually an option string and an option message.

```
options:add-unique-option 'h'
"print this help message"
options:add-unique-option 'v'
"print system version"
options:add-vector-option 'i'
"add a resolver path"
options:add-string-option 'e'
"force the encoding mode"
options:add-list-option 'f' "assert"
"enable assertion checks"
options:add-list-option 'f' "nopath"
"do not set initial path"
```

The above example shows the option descriptors for the interpreter. Since `[i]` is a vector option, multiple occurrences of that option is allowed. It shall be noted that the list option `[f assert]` is a debug option. This means that this option is always set when the program is compiled in debug mode.

4.3. Options parsing and retrieval. A string vector is parsed with the *parse* method. Generally, the vector argument is the interpreter argument vector defined in the qualified name *interp:args* . When the vector has been successfully parsed, it is possible to check the option that have been set.

```
options:parse (Vector "-h")
if (options:get-unique-option 'h') {
  options:usage
  afnix:sys:exit 0
}
```

In the above example, the option vector is parsed with the *parse* method. The *get-unique-option* method returns true for the `[h]` thus triggering the display of the usage message.

```
usage: axi [options] [file [arguments]]
[h] print this help message
[v] print system version
[i path] add a resolver path
[e mode] force the encoding mode
[f assert] enable assertion checks
[f nopath] do not set initial path
```

If the option is a string option, the *get-string-option* will return the string associated with that option. It shall be noted that the *get-unique-option* method can be used to check if the option has been set during the parsing process. If the option is a vector option, the *get-vector-option* method is more appropriate. In this case, a vector is returned with all strings matching this option.

```
1 options:parse (
2   Vector "-i" "../" "-i" "../.." -e "UTF-08" "hello")
```

In the previous example, the vector option `[i]` is set two times. The associated vector option has therefore a length of 2. The string option `[e]` is set to `UTF-08` . For this option `[e]` , the *get-unique-option* method will return true. Finally, the vector argument is filled with one string argument.

CHAPTER 18

Standard System Access Reference

1. Object Time

The *Time* class is a simple class used to manipulate time. The **AFNIX** system operates with a special coordinated time which uses the reference of Jan 1st 0000 in a modified *proleptic gregorian calendar* . Note that the time can be negative. Although, the class provides several methods to access the time fields, it is also possible to get a string representation that conforms to ISO-8601 or to RFC-2822. The resolution is in seconds. With 1 argument, the object is initialized with the time clock specified as an integer argument. With 3 arguments, the time is expressed with its different elements.

1.1. Predicate.

- time-p

1.2. Inheritance.

- Object

1.3. Constructors.

- *Time → (none)*
  The *Time* constructor create a time object which is initialized with the current time.
- *Time → (Integer)*
  The *Time* constructor create a time object which is initialized with the time argument.
- *Time → (Integer Integer Integer)*
  The *Time* constructor create a time object which is initialized with the time specific arguments, which are the hour, the minutes and the seconds.

1.4. Methods.

- *add → none (Integer)*
  The *add* method adds the time argument in seconds to the current time value This method is useful to compute a time in the future, in reference to the current time.

- *add-minutes → none (Integer)*
  The *add-minutes* method adds one or several minutes to the current time value. This method is useful to compute a time in the future, in reference to the current time.
- *add-hours → none (Integer)*
  The *add-hour* method adds one or several hours to the current time value. This method is useful to compute a time in the future, in reference to the current time.

- *add-days → none (Integer)*
  The *add-days* method adds one or several days to the current time value. This method is useful to compute a time in the future, in reference to the current time.

- *set-time → none (Integer)*
  The *set-time* method set the absolute time in seconds.
- *get-time → Integer (none—Boolean)*
  The *get-time* method returns absolute time in seconds. Without argument, the absolute time is computed in reference to the UTC time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.

- *seconds → Integer (none—Boolean)*
  The *seconds* method returns the number of seconds after the minute. Without argument, the number of seconds is computed in reference to the UTC time. With a boolean argument set to *true* , the number of seconds is computed in reference to the UTC time. If the argument is *false* , the local time is used. The returned value is the range 0 to 60.
- *minutes → Integer (none—Boolean)*
  The *minutes* method returns the number of minutes after the hour. Without argument, the number of minutes is computed in reference to the UTC time. With a boolean argument set to *true* , the number of minutes is computed in reference to the UTC time. If the argument is *false* , the local time is used. The returned value is the range 0 to 60.
- *hours → Integer (none—Boolean)*
  The *hours* method returns the number of hours since midnight. Without argument, the number of hours is computed in reference to the local time. With a boolean argument set to *true* , the number of hours is computed in reference to the UTC time. If the argument is *false* , the local time is used. The returned value is the range 0 to 23.
- *format → String (none—Boolean)*
  The *format* method returns a formatted representation of the time in the form of *hh:mm:ss* . Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-iso → String (none—Boolean)*
  The *to-iso* method returns a formatted representation of the time as specified by ISO-8601. Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-rfc → String (none—Boolean)*
  The *to-rfc* method returns a formatted representation of the time as specified by RFC-2822. Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *get-base-day → Integer (none)*
  The *get-base-day* method returns the absolute time rounded to the beginning of the day.

2. Object Date

The *Date* is a derived class designed to manipulate dates. The date computation is based on an *modified proleptic gregorian* calendar. This proleptic feature means that the actual calendar (gregorian) is extended beyond year 1582 (its introduction year) and modified in order to support the year 0. This kind of calendar is somehow similar to the astronomical gregorian calendar except that the reference date is 0 for special coordinated time. This method presents the advantage to support negative time. It should be noted that the 0 reference does not means year 1BC since year 0 did not exist at that time (the concept of zero is fairly new) and more important, the date expressed in the form 1BC generally refers to the Julian calendar since the date is before 1582. Although, the class provides several methods to access the individual fields, it is also possible to get a string representation that conforms to ISO-8601 or to RFC-2822. With 1 argument, the date is initialized with the time clock specified as an integer argument. With 3 or 6 arguments, the date is expressed with its different elements.

2.1. Predicate.

- date-p

2.2. Inheritance.

- Time

2.3. Constructors.

- *Date → (none)*
  The *Date* constructor creates a date object which is initialized with the current time.
- *date → (Integer)*
  The *Date* constructor creates a date object which is initialized with the time argument.
- *Date → (Integer Integer Integer)*
  The *Date* constructor creates a date object which is initialized with the date specific arguments, which are the year, the month and the day in the month.
- *Date → (Integer Integer Integer Integer Integer Integer)*
  The *Date* constructor creates a date object which is initialized with the date specific arguments, which are the year, the month, the day in the month, the hours, the minutes and the seconds.

2.4. Methods.

- *year → Integer (none—Boolean)*
  The *year* method returns the date year. the returned value is an absolute year value which can be negative. Without argument, the number of years is computed in reference to the local time. With a boolean argument set to *true* , the number of years is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *month → Integer (none—Boolean)*
  The *month* method returns the month in the year. The returned value is the range 1 to 12. Without argument, the number of months is computed in reference to the local time. With a boolean argument set to *true* , the number of months is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *day → Integer (none—Boolean)*
  The *day* method returns the day in the month. The returned value is the range 1 to 31. Without argument, the number of days is computed in reference to the

local time. With a boolean argument set to *true* , the number of days is computed in reference to the UTC time. If the argument is *false* , the local time is used.

- *week-day → Integer (none—Boolean)*
  The *week-day* method returns the day in the week. The returned value is the range 0 to 6 in reference to Sunday. Without argument, the day is computed in reference to the local time. With a boolean argument set to *true* , the day is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *year-day → Integer (none—Boolean)*
  The *year-day* method returns the day in the year. The returned value is the range 1 to 366 in reference to January 1st. Without argument, the day is computed in reference to the local time. With a boolean argument set to *true* , the day is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *map-day → String (none—Boolean)*
  The *map-day* method returns a formatted representation of the day. Without argument, the day is computed in reference to the local time. With a boolean argument set to *true* , the day is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *map-month → String (none—Boolean)*
  The *map-month* method returns a formatted representation of the month. Without argument, the month is computed in reference to the local time. With a boolean argument set to *true* , the month is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *format → String (none—Boolean)*
  The *format* method returns a formatted representation of the date. Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-iso → String (none—Boolean)*
  The *to-iso* method returns a formatted representation of the date as specified by ISO-8601. Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-web → String (none)*
  The *to-web* method returns a formatted representation of the date as specified by RFC-1123.
- *to-rfc → String (none—Boolean)*
  The *to-rfc* method returns a formatted representation of the date as specified by RFC-2822. Without argument, the time is computed in reference to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-date → String (none—Boolean)*
  The *to-date* method returns a formatted representation of the date only as specified by ISO-8601. With this method, the time value is not included in the representation. Without argument, the date is computed in reference to the local time. With a boolean argument set to *true* , the date is computed in reference to the UTC time. If the argument is *false* , the local time is used.
- *to-time → String (none—Boolean)*
  The *to-time* method returns a formatted representation of the time as returned by the *Time format* method. Without argument, the time is computed in reference

to the local time. With a boolean argument set to *true* , the time is computed in reference to the UTC time. If the argument is *false* , the local time is used.

- *add-years → none (Integer)*
  The *add-years* method add one or several years to the current date.
- *add-months → none (Integer)*
  The *add-months* method add one or several months to the current date.

3. Object Options

The *Options* class is a simple class used to define and retrieve user options. The object is constructed by specifying which option is valid and how it behaves. The arguments can be passed to the object for subsequent analysis. An option can be either a unique option or a string option. In this later case, multiple value for the same option can be accepted. In that case, the option is said to be a string vector option. An option can be also an option list. I that case, the option is defined with a set of valid string. A list option is associated with a boolean flag for each string defined with that option.

3.1. Predicate.

- options-p

3.2. Inheritance.

- Object

3.3. Constructors.

- *Options → (none)*
  The *Options* constructor creates a default option object without a user message.
- *Options → (String)*
  The *Options* constructor creates an empty option object with a user message. The user message is used by the *usage* method.

3.4. Methods.

- *reset → none (none)*
  The *reset* method resets the object data structure but do not remove the option descriptors. After a reset operation, the class is ready to parse another string vector.
- *usage → none (none)*
  The *usage* method prints a usage message with a user message and a one line description per option. removing all messages.
- *parse → Vector (none)*
  The *parse* method parse a vector and fill the option data structure. The parse method is generally called with the interpreter argument vector.
- *empty-p → Boolean (none)*
  The *empty-* predicate returns true if the argument vector is empty. The argument vector is filled wit the string that are not options during the parsing process.
- *add-list-option → none (Character String String)*
  The *add-list-option* method creates a new list option. The list option is defined by the option character and the option string. The first argument is the option character. The second argument is the option list string. The third argument is the option message. During the parsing process, the list option have a string argument which must match one string associated with the option character.
- *get-unique-option → Character String (none)*
  The *add-unique-option* method creates a new single option. The option is defined only by its character. The first argument is the option character. The second argument is the option message. During the parsing process, a unique option does not have an argument.
- *add-string-option → none (Character String)*
  The *add-string-option* method creates a new string option. The option is defined only by its character. The first argument is the option character. The second argument is the option message. During the parsing process, a string option have a string argument.

- *add-vector-option → Character String (none)*
  The *add-vector-option* method creates a new vector option. The option is defined only by its character. The first argument is the option character. The second argument is the option message. During the parsing process, a vector option have a string argument which is accumulated in a vector.
- *set-user-message → none (String)*
  The *set-user-message* method sets the global option user message. The user message is used by the *usage* method.
- *get-user-message → String (none)*
  The *get-user-message* method returns the global option user message. The user message is used by the *usage* method.
- *get-unique-option → Boolean (Character)*
  The *get-unique-option* method returns the flag associated with an option. If the option has been detected during the parsing process, the method returns true. This method works also for string option or list option to indicate if the string has been set for that option. with a vector option, it is simpler to get the vector and check for the vector length. The first argument is the option character to use for testing.
- *get-string-option → String (Character)*
  The *get-string-option* method returns the string associated with a string option. In order to make sure that a string option has been properly set during the parsing process, it is recommended to use the *get-unique-option* method. The first argument is the option character to use for the string retrieval.
- *get-vector-option → Vector (Character)*
  The *get-vector-option* method returns the vector associated with a vector option. The first argument is the option character to use for the vector retrieval.
- *get-vector-arguments → Vector (none)*
  The *get-vector-arguments* method returns the vector arguments built during the parsing process.

3.5. Functions.

- *exit → none (Integer)*
  The *exit* function terminates the executing program with the exit code specified as the argument.
- *sleep → none (Integer)*
  The *sleep* function pause the specific thread for a certain time. The time argument is expressed in milliseconds. This function returns nil.
- *get-option → String (Character)*
  The *get-option* function returns a formatted string equivalent to the system option as specified by the character argument.
- *get-unique-id → Integer (none)*
  The *get-unique-id* function returns an unique integer number. The returned number is unique across the session.
- *get-pid → Integer (none)*
  The *get-pid* function returns the process identifier (pid). The returned value is a positive integer.
- *get-env → String (String)*
  The *get-env* function returns the environment variable associated with the string argument. If the environment does not exist an exception is raised.
- *get-host-fqdn → String (none)*
  The *get-host-fqdn* function returns the host fully qualified domain name. This

is the combined host and domain names which is sometimes called the canonical name.

- *get-domain-name → String (none)*
  The *get-domain-name* function returns the host domain name.
- *get-host-name → String (none)*
  The *get-host-name* function returns the host name. If the host does not have a domain name, the host name is equal to the fully qualified domain name.
- *get-user-name → String (none)*
  The *get-user-name* function returns the current user name.

CHAPTER 19

Standard Text Processing Module

The *Standard Text Processing* module is an original implementation of an object collection dedicated to text processing. Although text scaning is the current operation perfomed in the field of text processing, the module provides also specialized object to store and index text data. Text sorting and transliteration is also part of this module.

1. Scanning concepts

Text scanning is the ability to extract lexical elements or *lexemes* from a stream. A scanner or lexical analyzer is the principal object used to perform this task. A scanner is created by adding special object that acts as a pattern matcher. When a pattern is matched, a special object called a *lexeme* is returned.

1.1. Pattern object. A *Pattern* object is a special object that acts as model for the string to match. There are several ways to build a pattern. The simplest way to build it is with a regular expression. Another type of pattern is a balanced pattern. In its first form, a pattern object can be created with a regular expression object.

```
# create a pattern object
const pat (afnix:txt:Pattern "$d+")
```

In this example, the pattern object is built to detect integer objects.

```
pat:check "123" # true
pat:match "123" # 123
```

The *check* method return true if the input string matches the pattern. The *match* method returns the string that matches the pattern. Since the pattern object can also operates with stream object, the *match* method is appropriate to match a particular string. The pattern object is, as usual, available with the appropriate predicate.

```
afnix:txt:pattern-p pat # true
```

Another form of pattern object is the balanced pattern. A balanced pattern is determined by a starting string and an ending string. There are two types of balanced pattern. One is a single balanced pattern and the other one is the recursive balanced pattern. The single balanced pattern is appropriate for those lexical element that are defined by a character. For example, the classical C-string is a single balanced pattern with the double quote character.

```
# create a balanced pattern
const pat (afnix:txt:Pattern "ELEMENT" "<" ">")
pat:check "<xml>" # true
pat:match "<xml>" # xml
```

In the case of the C-string, the pattern might be more appropriately defined with an additional *escape character* . Such character is used by the pattern matcher to grab characters that might be part of the pattern definition.

```
# create a balanced pattern
const pat (afnix:txt:Pattern "STRING" "'" '\\')
pat:check "'hello'" # true
pat:match "'hello'" # "hello"
```

In this form, a balanced pattern with an escape character is created. The same string is used for both the starting and ending string. Another constructor that takes two strings can be used if the starting and ending strings are different. The last pattern form is the balanced recursive form. In this form, a starting and ending string are used to delimit the pattern. However, in this mode, a recursive use of the starting and ending strings is allowed. In order to have an exact match, the number of starting string must equal the number of ending string. For example, the C-comment pattern can be viewed as recursive balanced pattern.

```
# create a c-comment pattern
const pat (afnix:txt:Pattern "STRING" "/*" "*/" )
```

1.2. Lexeme object. The *Lexeme* object is the object built by a scanner that contains the matched string. A lexeme is therefore a tagged string. Additionally, a lexeme can carry additional information like a source name and index.

```
# create an empty lexeme
const lexm (afnix:txt:Lexeme)
afnix:txt:lexeme-p lexm # true
```

The default lexeme is created with any value. A value can be set with the *set-value* method and retrieved with the *get-value* methods.

```
lexm:set-value "hello"
lexm:get-value # hello
```

Similar are the *set-tag* and *get-tag* methods which operate with an integer. The source name and index are defined as well with the same methods.

```
# check for the source
lexm:set-source "world"
lexm:get-source # world
# check for the source index
lexm:set-index 2000
lexm:get-index # 2000
```

2. Text scanning

Text scanning is the ability to extract lexical elements or lexemes from an input stream. Generally, the lexemes are the results of a matching operation which is defined by a pattern object. As a result, the definition of a scanner object is the object itself plus one or several pattern object.

2.1. Scanner construction. By default, a scanner is created without pattern objects. The *length* method returns the number of pattern objects. As usual, a predicate is associated with the scanner object.

```
# the default scanner
const scan (afnix:txt:Scanner)
afnix:txt:scanner-p scan # true
# the length method
scan:length # 0
```

The scanner construction proceeds by adding pattern objects. Each pattern can be created independently, and later added to the scanner. For example, a scanner that reads real, integer and string can be defined as follow:

```
# create the scanner pattern
const REAL (
  afnix:txt:Pattern "REAL" [$d+.$d*])
const STRING (
  afnix:txt:Pattern "STRING" """ '\\')
const INTEGER (
  afnix:txt:Pattern "INTEGER" [$d+|"0x"$x+])
# add the pattern to the scanner
scanner:add INTEGER REAL STRING
```

The order of pattern integration defines the priority at which a token is recognized. The symbol name for each pattern is optional since the functional programming permits the creation of patterns directly. This writing style makes the scanner definition easier to read.

2.2. Using the scanner. Once constructed, the scanner can be used *as is* . A stream is generally the best way to operate. If the scanner reaches the end-of-stream or cannot recognize a lexeme, the nil object is returned. With a loop, it is easy to get all lexemes.

```
while (trans valid (is:valid-p)) {
  # try to get the lexeme
  trans lexm (scanner:scan is)
  # check for nil lexeme and print the value
  if (not (nil-p lexm)) (println (lexm:get-value))
  # update the valid flag
  valid:= (and (is:valid-p) (not (nil-p lexm)))
}
```

In this loop, it is necessary first to check for the end of the stream. This is done with the help of the special loop construct that initialize the *valid* symbol. As soon as the the lexeme is built, it can be used. The lexeme holds the value as well as it tag.

3. Text sorting

Sorting is one the primary function implemented inside the *text processing* module. There are three sorting functions available in the module.

3.1. Ascending and descending order sorting. The *sort-ascent* function operates with a vector object and sorts the elements in ascending order. Any kind of objects can be sorted as long as they support a comparison method. The elements are sorted in placed by using a *quick sort* algorithm.

```
# create an unsorted vector
const v-i (Vector 7 5 3 4 1 8 0 9 2 6)
# sort the vector in place
afnix:txt:sort-ascent v-i
# print the vector
for (e) (v) (println e)
```

The *sort-descent* function is similar to the *sort-ascent* function except that the object are sorted in descending order.

3.2. Lexical sorting. The *sort-lexical* function operates with a vector object and sorts the elements in ascending order using a lexicographic ordering relation. Objects in the vector must be literal objects or an exception is raised.

4. Transliteration

Transliteration is the process of changing characters my mapping one to another one. The transliteration process operates with a character source and produces a target character with the help of a mapping table. The transliteration process is not necessarily reversible as often indicated in the literature.

4.1. Literate object. The *Literate* object is a transliteration object that is bound by default with the identity function mapping. As usual, a predicate is associate with the object.

```
# create a transliterate object
const tl (afnix:txt:Literate)
# check the object
afnix:txt:literate-p tl # true
```

The transliteration process can also operate with an escape character in order to map double character sequence into a single one, as usually found inside programming language.

```
# create a transliterate object by escape
const tl (afnix:txt:Literate '\\')
```

4.2. Transliteration configuration. The *set-map* configures the transliteration mapping table while the *set-escape-map* configure the escape mapping table. The mapping is done by setting the source character and the target character. For instance, if one want to map the tabulation character to a white space, the mapping table is set as follow:

```
tl:set-map '\t' ' '
```

The escape mapping table operates the same way. It should be noted that the mapping algorithm translate first the input character, eventually yielding to an escape character and then the escape mapping takes place. Note also that the *set-escape* method can be used to set the escape character.

```
tl:set-map '\t' ' '
```

4.3. Transliteration process. The transliteration process is done either with a string or an input stream. In the first case, the *translate* method operates with a string and returns a translated string. On the other hand, the *read* method returns a character when operating with a stream.

```
# set the mapping characters
tl:set-map '\h' 'w'
tl:set-map '\e' 'o'
tl:set-map '\l' 'r'
tl:set-map '\o' 'd'
# translate a string
tl:translate "helo" # word
```

CHAPTER 20

Standard Text Processing Reference

1. Object Pattern

The *Pattern* class is a pattern matching class based either on regular expression or balanced string. In the regex mode, the pattern is defined with a regex and a matching is said to occur when a regex match is achieved. In the balanced string mode, the pattern is defined with a start pattern and end pattern strings. The balanced mode can be a single or recursive. Additionally, an escape character can be associated with the class. A name and a tag is also bound to the pattern object as a mean to ease the integration within a scanner.

1.1. Predicate.

- pattern-p

1.2. Inheritance.

- Object

1.3. Constructors.

- *Pattern → (none)*
  The *Pattern* constructor creates an empty pattern.
- *Pattern → (String—Regex)*
  The *Pattern* constructor creates a pattern object associated with a regular expression. The argument can be either a string or a regular expression object. If the argument is a string, it is converted into a regular expression object.
- *Pattern → (String String)*
  The *Pattern* constructor creates a balanced pattern. The first argument is the start pattern string. The second argument is the end balanced string.
- *Pattern → (String String Character)*
  The *Pattern* constructor creates a balanced pattern with an *escape character* . The first argument is the start pattern string. The second argument is the end balanced string. The third character is the *escape character* .
- *Pattern → (String String Boolean)*
  The *Pattern* constructor creates a recursive balanced pattern. The first argument is the start pattern string. The second argument is the end balanced string.

1.4. Constants.

- *REGEX → ()*
  The *REGEX* constant indicates that the pattern is a regular expression.
- *BALANCED → ()*
  The *BALANCED* constant indicates that the pattern is a balanced pattern.
- *RECURSIVE → ()*
  The *RECURSIVE* constant indicates that the pattern is a recursive balanced pattern.

1.5. Methods.

- *check → Boolean (String)*
  The *check* method checks the pattern against the input string. If the verification is successful, the method returns true, false otherwise.
- *match → String (String—InputStream)*
  The *match* method attempts to match an input string or an input stream. If the matching occurs, the matching string is returned. If the input is a string, the end of string is used as an end condition. If the input stream is used, the end of stream is used as an end condition.

- *set-tag → none (Integer)*
  The *set-tag* method sets the pattern tag. The tag can be further used inside a scanner.
- *get-tag → Integer (none)*
  The *get-tag* method returns the pattern tag.
- *set-name → none (String)*
  The *set-name* method sets the pattern name. The name is symbol identifier for that pattern.
- *get-name → String (none)*
  The *get-name* method returns the pattern name.
- *set-regex → none (String—Regex)*
  The *set-regex* method sets the pattern regex either with a string or with a regex object. If the method is successfully completed, the pattern type is switched to the REGEX type.
- *set-escape → none (Character)*
  The *set-escape* method sets the pattern escape character. The escape character is used only in balanced mode.
- *get-escape → Character (none)*
  The *get-escape* method returns the escape character.
- *set-balanced → none (String— String String)*
  The *set-balanced* method sets the pattern balanced string. With one argument, the same balanced string is used for starting and ending. With two arguments, the first argument is the starting string and the second is the ending string.

2. Object Lexeme

The *Lexeme* class is a literal object that is designed to hold a matching pattern. A lexeme consists in string (i.e. the lexeme value), a tag and eventually a source name (i.e. file name) and a source index (line number).

2.1. Predicate.

- lexeme-p

2.2. Inheritance.

- Literal

2.3. Constructors.

- *Lexeme → (none)*
  The *Lexeme* constructor creates an empty lexeme.
- *Lexeme → (String)*
  The *Lexeme* constructor creates a lexeme by value. The string argument is the lexeme value.

2.4. Methods.

- *set-tag → none (Integer)*
  The *set-tag* method sets the lexeme tag. The tag can be further used inside a scanner.
- *get-tag → Integer (none)*
  The *get-tag* method returns the lexeme tag.
- *set-value → none (String)*
  The *set-value* method sets the lexeme value. The lexeme value is generally the result of a matching operation.
- *get-value → String (none)*
  The *get-value* method returns the lexeme value.
- *set-index → none (Integer)*
  The *set-index* method sets the lexeme source index. The lexeme source index can be for instance the source line number.
- *get-index → Integer (none)*
  The *get-index* method returns the lexeme source index.
- *set-source → none (String)*
  The *set-source* method sets the lexeme source name. The lexeme source name can be for instance the source file name.
- *get-source → String (none)*
  The *get-source* method returns the lexeme source name.

3. Object Scanner

The *Scanner* class is a text scanner or *lexical analyzer* that operates on an input stream and permits to match one or several patterns. The scanner is built by adding patterns to the scanner object. With an input stream, the scanner object attempts to build a buffer that match at least one pattern. When such matching occurs, a lexeme is built. When building a lexeme, the pattern tag is used to mark the lexeme.

3.1. Predicate.

- scanner-p

3.2. Inheritance.

- Object

3.3. Constructors.

- *Scanner → (none)*
  The *Scanner* constructor creates an empty scanner.

3.4. Methods.

- *add → none (Pattern*)*
  The *add* method adds 0 or more pattern objects to the scanner. The priority of the pattern is determined by the order in which the patterns are added.
- *length → Integer (none)*
  The *length* method returns the number of pattern objects in this scanner.
- *get → Pattern (Integer)*
  The *get* method returns a pattern object by index.
- *check → Lexeme (String)*
  The *check* method checks that a string is matched by the scanner and returns the associated lexeme.
- *scan → Lexeme (InputStream)*
  The *scan* method scans an input stream until a pattern is matched. When a matching occurs, the associated lexeme is returned.

4. Object Literate

The *Literate* class is transliteration mapping class. Transliteration is the process of changing characters my mapping one to another one. The transliteration process operates with a character source and produces a target character with the help of a mapping table. This transliteration object can also operate with an escape table. In the presence of an escape character, an escape mapping table is used instead of the regular one.

4.1. Predicate.

- literate-p

4.2. Inheritance.

- Object

4.3. Constructors.

- *Literate → (none)*
  The *Literate* constructor creates a default transliteration object.
- *Literate → (Character)*
  The *Literate* constructor creates a default transliteration object with an *escape character* . The argument is the escape character.

4.4. Methods.

- *read → Character (InputStream)*
  The *read* method reads a character from the input stream and translate it with the help of the mapping table. A second character might be consumed from the stream if the first character is an escape character.
- *getu → Character (InputStream)*
  The *getu* method reads a Unicode character from the input stream and translate it with the help of the mapping table. A second character might be consumed from the stream if the first character is an escape character.
- *reset → none (none)*
  The *reset* method resets all the mapping table and install a default identity one.
- *set-map → none (Character Character)*
  The *set-map* method set the mapping table by using a source and target character. The first character is the source character. The second character is the target character.
- *get-map → Character (Character)*
  The *get-map* method returns the mapping character by character. The source character is the argument.
- *translate → String (String)*
  The *translate* method translate a string by transliteration and returns a new string.

- *set-escape → none (Character)*
  The *set-escape* method set the escape character.
- *get-escape → Character (none)*
  The *get-escape* method returns the escape character.
- *set-escape-map → none (Character Character)*
  The *set-escape-map* method set the escape mapping table by using a source and target character. The first character is the source character. The second character is the target character.
- *get-escape-map → Character (Character)*
  The *get-escape-map* method returns the escape mapping character by character. The source character is the argument.

4.5. Functions.

- *sort-ascent → none (Vector)*
  The *sort-ascent* function sorts in ascending order the vector argument. The vector is sorted in place.
- *sort-descent → none (Vector)*
  The *sort-descent* function sorts in descending order the vector argument. The vector is sorted in place.
- *sort-lexical → none (Vector)*
  The *sort-lexical* function sorts in lexicographic order the vector argument. The vector is sorted in place.

CHAPTER 21

Standard XML Module

The *Standard XML* module is an original implementation of the XML markup language. The module provides the necessary objects for parsing a xml description as well as manipulating the parsed tree. The module can be extended to a service as a mean to act as a XML processor. The module also provides the support for a *simple model* which enable the quick parsing of documents with a relaxed rule checking approach.

1. XML tree representation

A xml document is represented with a tree. At the top of the tree is the *XmlRoot* object. The root object is not part of the document, but acts as the primary container for other objects. A xml document starts with a root node and all other child elements are *XmlNode* objects.

1.1. Node base object. The xml tree is built with the *XmlNode* object. The node object has different derivation depending on the required representation. For example, the *XmlRoot* object is derived from the *XmlNode* object. A node object can have child object unless the node is marked as an *empty node* . Trying to add node to an empty node results in an exception. A node can also be marked empty by the user. This situation typically arises with tag node which are used alone such like the *
* xhtml empty tag or an empty paragraph *<p/>* . Although a xml node cannot be constructed directly, there is a predicate *node-p* that can be used to assert the node type.

```
1 # check a node
2 assert true (afnix:xml:node-p node)
```

The *add-child* method adds a child node to the calling node. If the calling node is marked empty, an exception is raised when attempting to add the node. There is no limit for the number of nodes to add. In particular, when a text is to be added, care should be taken that there is no markup within that text. In doubt, the *parse* method should be used.

```
1 # parse a text and add 3 child nodes
2 p:parse "The quick brown <b>fox</b>
3 jumps over the lazy dog"
```

In the previous example, the first child node is a *XmlText* node with the value *The quick brown* . The second node is a *XmlTag* node with name *b* . Finally, the third node is also a *XmlText* node with the value *jumps over the lazy dog* . It should be noted that the tag node has a child *XmlText* node with the value *fox* . This example also illustrates the power of the *parse* method which considerably simplify the creation of a xml tree. Finally, there is a subtle subject to be treated later which concerns the use of *character reference* with the *parse* method. Like any other xml parser, character references are evaluated during the parsing phase, thus providing no mechanism to create such reference. For this reason, a special class called *XmlCref* is provided in the module.

1.2. Tag object. The *XmlTag* class is one of the most important class as it holds most of the xml constructs. A tag is defined by a name, a set of attributes and eventually a content. In its simplest form, a tag is created by name. With an additional boolean parameter, the tag can be marked as an empty node.

```
# create an empty paragraph tag
const p (afnix:xml:XmlTag "p" true)
```

Adding attributes to a tag is imply a matter of method call. The *add-attribute* method operates with a *Property* object while the *set-attribute* operates with a name and a literal value. As a matter of fact, the attributes are stored internally as a property list.

```
# <p class="text">
# create a paragraph tag
const p (afnix:xml:XmlTag "p")
# set the class attribute
p:set-attribute "class" "text"
```

The node empty flag determines whether or not there is a end tag associated with a tag. If the empty flag is false, the node can have children nodes and is associated with a end tag. With the empty flag set, there is no child nodes. Such situation corresponds to the xml /> notation.

```
# <br/>
# create a br empty tag
const br (afnix:xml:XmlTag "br" true)
```

1.3. Text objects. The xml module provides two types of xml text node. The basic object is the *XmlText* node which is designed to hold some text without markup. It is this kind of nodes which is automatically instantiated by the *parse* method, as described earlier. The other object is the *XmlData* which corresponds to the xml *CDATA* special markup. With a character data node, the characters are not interpreted, including those that indicate markup starts like < or end like > . The *XmlData* is particularly used to store scripts or other *program text* inside a xml description. As an example, it is recommended to use a character data node inside a script tag with xhtml.

2. Document reading

A xml document is read by scanning an input stream an building a representation of the xml tree.

2.1. The document object. The *XmlDocument* object is a special object is designed to ease the reading process of an xml document. The process of creating a xml document consists of creating a document object, then binding a xml reader, parsing the input stream and finally storing the root node. When the operation is completed, the root node is available in the document object.

```
# create a xml document
const xdoc (afnix:xml:XmlDocument "example.xml")
# get the root node
const rppt (xdoc:get-root)
```

2.2. The root node content. When a document is parsed, the root node holds all the elements and markup sequentially. At this stage, it shall be noted that the element data are not expanded. Unlike a normal XML reader, the parameter entity are kept in the node data, are expended later by the XML processor.

3. Node tree operations

The class *XneTree* provides a single framework to operate on a node and its associated tree. Since a node always carries a sub-tree, the *node tree* term will be used to reference it.

3.1. Creating a node tree. A node tree is created either from a node at construction or with the help of the *set-node* method.

```
# create a node tree at construction
const tree (afnix:xml:XneTree root)
# change the node tree
tree:set-node node
```

Once a tree is created, various methods are provided to operate on the whole tree. The *depth* method returns the depth of the node tree. The *get-node* methods returns the the node associated with the tree.

```
# get the tree depth
println (tree:depth)
```

3.2. Namespace operations. The concept of *namespace* is an extension to the xml standard. Unlike other programming language, the concept of namespace is designed to establish a binding between a name and an uri. Such binding permits to establish a scope for tags without too much burden. In the xml namespace terminology, an *expanded name* is composed of a *prefix* and a *local name* . The basic operations provided at the tree level is the prefix cancellation and the tree prefix setting.

```
# clear the prefix for the whole tree
tree:clear-prefix
# set a prefix for the whole tree
tree:set-prefix "afnix"
```

The *set-prefix* changes the prefix for the whole tree. It is not necessary to clear first the prefix.

3.3. Attribute operations. Each node in the node tree can have its attribute list modified in a single operation. The first operation is to clear all attributes for all nodes. Although this operation might be useful, it should be carried with caution. The attributes can also cleared more selectively by using the tag name as a filter. For more complex operation, the *clear-attribute* method of the *XmlTag* is the definitive answer.

```
# clear all attributes
tree:clear-attribute
# clear all attributes by tag name
tree:clear-attribute "p"
```

The *set-attribute* method sets an attribute to the whole tree. The first argument is the attribute name and the second is a literal value. For more selective operations, the *set-attribute* method can be also called at the tag level.

```
# clear all attributes
tree:set-attribute "class" "text"
```

When it comes to set attributes, there is a special operation related to the ”id” attribute. Such attribute is supposed to be unique for the whole tree. For this reason, the *generate-id* generates a unique id for each node and assign the id attribute. The attribute is unique at the time of the call. If the tree is modified, and in particular, if new node are added, the method must be called again to regenerate the node id.

```
# set a unique id for all nodes
tree:generate-id
```

4. Node location and searching

The node location is the ability to locate one or several nodes in a xml tree. A node is generally located by name, path or id. Once a node has been located, it can be processed. Note that the node locator operates operates almost exclusively with *XmlTag* node, although it might not be always the case.

4.1. Node selection. The process of finding a child node is obtained with the help of the *XneCond* class combined with the *select* method of the *XneTree* Object. The *select* method traverses the whole tree and attempts to match a condition for each node. If the condition is evaluated successfully for a node, the node is added in the result vector. Note that the tree can be traversed entirely or with only the first layer of children.

```
# creating a condition node
const xcnd (afnix:xml:XneCond)
# create a tree with a root node
const tree (afnix:xml:XneTree root)
# select all nodes for that condition
trans result (tree:select xcnd)
```

In the previous example, the condition object is empty. This means that there is no condition, and thus works for all nodes. This previous example will return all nodes in the tree.

4.2. Node condition. The *XmlCond* class provides several method to add a conditions. The *add* method is the method of choice to add a condition. The method operates with a condition type and a literal. Note that the object can contain several conditions.

```
# creating a condition node
const xcnd (afnix:xml:XneCond)
# add a condition by name
xcnd:add afnix:xml:xne:NAME "p"
```

In the previous example, a condition is designed to operate with a tag name. Upon a call to the *select* method with this condition, all nodes in the tree that have the tag name *p* will be selected.

```
# creating a condition node
const xcnd (afnix:xml:XneCond)
# add a condition by name
xcnd:add afnix:xml:xne:NAME "p"
# add an index condition
xcnd:add afnix:xml:xne:INDEX 0
```

In the previous example, a condition is designed to operate with a tag name and index. Upon a call to the *select* method with this condition, all nodes in the tree that have the tag name *p* and those child index is 0 will be selected.

4.3. Selection result. The node selection operates by default on the whole tree. The *select* method, when called with a second boolean argument can restrict the search to the child nodes.

```
# creating a condition node
const xcnd (afnix:xml:XneCond)
# create a tree with a root node
const tree (afnix:xml:XneTree root)
# select all nodes for that condition
trans result (tree:select xcnd false)
```

The selection results is stored in a vector object. The node order corresponds to the tree order obtained with a depth first search approach.

5. Simple model node

The XML simple model is designed to simplify the interpretation of a general sgml document such like, html or xhtml document. In the simple model approach, there is no tree. Instead, a vector of simple nodes is built, and a document interface can be used to access the nodes. Therefore, this simple model should be considered as a mean to quickly parse document, but should not be used when tree operations come into play. In such case, the xml model is by far more appropriate. The simple model can be used to parse a html document for instance. Note also that the simple model is a relaxed model in terms of parsing rules. For example, the tag start/end consistency is not checked and the attribute parsing is not aggressive as it can be found generally in poorly written html document.

In the simple model, a *XsmNode* is just a text place holder. The node transports its type which can be either text, tag, reference of end node. For the tag node, a subtype that identifies reserved nodes versus normal type is also available.

5.1. Creating a node. A xsm node is created by name or byte and name. In the first case, the node is a text node. In the second case, the node subtype is automatically detected for tag node.

```
# create a xsm text node
const ntxt (afnix:xml:XsmNode "afnix">
  # create a xsm tag node
  const ntag (
    afnix:xml:XsmNode afnix:xml:XsmNode:TAG "afnix">
```

Note that the text corresponds to the node content. For example, the string "!– example –" might corresponds to a comment in html which is to say a reserved tag when the type is tag or a simple text if the type is a text node. A reserved tag is defined by a string which start either with the '!' character or the '[' character.

```
# create a reserved tag
const rtag (
  afnix:xml:XsmNode afnix:xml:XsmNode:TAG
  "!-- example --")
```

5.2. Node representation. The xsm node is a literal node. This means that the *to-string* and *to-literal* methods are available. When the *to-literal* method is called, the node text is automatically formatted to reflect the node type.

```
# create a reserved tag
const rtag (
  afnix:xml:XsmNode afnix:xml:XsmNode:TAG
  "!-- example --")
# print the node literal
rtag:to-literal # <!-- example -->
```

If the node is a reference node, the node literal is represented with the original definition while the *to-string* method will produce the corresponding character if it known.

5.3. Node information. With a xsm node, the operation are a limited number of *node information* operations. The *get-name* method returns the first name found in a node. If the node is a normal tag, the *get-name* will return the tag name. For the other node, the method will return the first available string. This also means, that the method will behave correctly with end tag node.

```
# create a tag node
const ntag (
  afnix:xml:XsmNode afnix:xml:XsmNode:TAG "afnix">
  # get the tag name
  ntag:get-name
```

There is a predicate for all types. For example, the *text-p* predicate returns true if the node is a text node. The *tag-p* predicate returns true if the node is a normal or reserved tag.

6. Document reading

A document is read in a way similar to the *XmlDocument* with the help of the *XsmDocument* object. Once created, the document holds a vector of nodes.

6.1. The document object. The *XsmDocument* object is a special xsm object designed to ease the reading process of a document. The process of creating a document consists of creating a document object, then binding a xsm reader, parsing the input stream and storing the nodes in a vector. When the operation is completed, the vector can be accessed by index.

```
# create a xms document
const xdoc (afnix:xml:XsmDocument "example.htm")
# get the document length
xdoc:length
```

6.2. Node information object. The *XsoInfo* object is a node information object designed to hold a node name, an attributes list and eventually a text associated with the node. For example, if a html document contains a anchor node, the associated information node, will have the anchoring text stored as the node information text.

```
# create a xso node by name and text
const info (afnix:xml:XsoInfo "a" "click here")
```

6.3. Simple model operations. The *XsmDocument* is designed to perform simple operations such like searching all nodes that matches a particular name. While this operation can be done easily, it is done in such a way that a vector of *node information* is returned instead of a vector of nodes which can always be constructed with a simple loop.

```
# create a xsm document
const xdoc (afnix:xml:XsmDocument "example.htm")
# get all node named "a" - forcing lower case
xdoc:get-info-vector "a" true
```

CHAPTER 22

Standard XML Reference

1. Object XmlNode

The *XmlNode* class is the base class used to represent the xml tree. The tree is built as a vector of nodes. Each node owns as well its parent node. Walking in the tree is achieved by taking the child node and then moving to the child and/or next node. The node also manages an empty flags. It the empty flag is set, it is an error to add child nodes.

1.1. Predicate.

- node-p

1.2. Inheritance.

- Object

1.3. Methods.

- *to-text → String (none)*
  The *to-text* method returns a text representation of the tree content. Unlike the write method, the tag are not generated, but rather the text content is accumulated. This method is useful tor read the node content. If a node does not have text, the nil string is returned.
- *write → none (none—OutputStream—Buffer)*
  The *write* method write the node contents as well as the child nodes to an output stream argument or a buffer. When node is written, the method attempts to use the stream encoding in such way that the contents fits into the requested output encoding. Without argument, the node is written to the interpreter output stream. with one argument, the node is written to the specified stream or buffer.
- *name-p → Boolean (String)*
  The *name-p* predicate checks if the name matches the node name. Care should be taken that not all node have a name, and in such case, the false value is returned. This method is useful when the node is a tag.
- *attribute-p → Boolean (String— String String)*
  The *attribute-p* predicate checks if there is a node attribute that matches the string argument name. In the first form, the predicate returns true is an attribute exists with the name argument. In the second form, the predicate returns true if the attribute name and value matches the arguments. The first argument is the attribute name. The second argument is the attribute value. Not all nodes have attributes. In such case, the predicate always returns false.
- *parse → none (String)*
  The *parse* method parses the string argument and adds the results as a set of child node to the calling node. If the node is an empty node, the method will almost fail. This method should be used when an attempt is made to add some text that may contain some xml tags.
- *get-parent → XmlNode (none)*
  The *get-parent* method returns the parent node. If the node is the root node, nil is returned.
- *set-parent → none (XmlNode)*
  The *set-parent* method sets the parent node.
- *copy → XmlNode (none)*
  The *copy* method copy the node tree by regenerating a new tree.
- *del-child → none (Integer — String — String String — String String String)*
  The *del-child* method deletes one or several child nodes. In the first form, the children is deleted either by index or by name. When a string argument is used, several node might be removed. In the second form, the child node name and

attribute name must be matched. In the third form, the child node name, attribute name and value must be matched.

- *del-attribute-child → none (String — String String)*
  The *del-attribute-child* method deletes one or several child nodes. In the first form, the children are deleted by attribute name. In the second form, the children are delete by attribute name and value.
- *clear-child → none (none)*
  The *clear-child* method clear the child node list, leaving the node without child node.
- *add-child → none (XmlNode—XmlNode Integer)*
  The *add-child* method adds a node argument as a child node to the calling node. In the first form, the node is added at the end of the node list. In the second form, the node is added by index and all subsequent nodes are shifted by one position.
- *get-child → XmlNode (Integer String)*
  The *get-child* method returns a child node by index or by name. If the calling argument is an integer, the node is returned by index. If the calling argument is a string, the node is returned by name. If the node cannot be found, nil is returned raised.
- *get-index → Integer (XmlNode)*
  The *gett-index* method returns a child node index. The node argument is the node to find as a child node. If the node is not found, an exception is raised.
- *merge → none (XmlNode Integer)*
  The *merge* method merge an existing node with another one. The first argument is the source node used for merging. The second argument the child node index to merge. The method operates by first removing the child node at the specified index and then add in position, the child nodes of the source node.
- *nil-child-p → Boolean (none)*
  The *nil-child-p* predicate returns true if the node does not have a child node.
- *child-p → Boolean (String — String String — String String String)*
  The *child-p* predicate returns true if the node has a child with a node name argument. In the first form, the name is to be matched by the predicate. In the second form, the node nae and the attribute name must be matched. In the third form, the node name, attribute name and value must be matched.
- *attribute-child-p → Boolean (String String — String String String)*
  The *attribute-child-p* predicate returns true if the node has a child with an attribute name argument. In the first form, the attribute name must be matched. In the second form, the attribute name and value must be matched.
- *lookup-child → XmlNode (String)*
  The *lookup-child* method returns a child node by name. Unlike the *get-child* method, the method raises an exception if the node cannot be found.
- *child-length → Integer (none—String)*
  The *child-length* method returns the number of children nodes. In the first form, without argument, the total number of children nodes is returned. In the second form, the total number of nodes that match the tag argument name is returned.
- *get-source-line → Integer (none)*
  The *get-source-line* method returns the node source line number if any.
- *set-source-line → none (Integer)*
  The *set-source-line* method sets the node source line number.
- *get-source-name → String (none)*
  The *get-source-name* method returns the node source name if any.

- *set-source-name → none (String)*
  The *set-source-name* method sets the node source name.

2. Object XmlTag

The *XmlTag* class is the base class used to represent a xml tag. A tag is defined with a name and an attribute list. The tag is derived from the xml node class and is not marked empty by default.

2.1. Predicate.

- tag-p

2.2. Inheritance.

- XmlNode

2.3. Constructors.

- *XmlTag → (String)*
  The *XmlTag* constructor creates a tag node. The node is not marked empty.
- *XmlTag → (String Boolean)*
  The *XmlTag* constructor creates a tag node. The first argument is the tag name. The second argument is the empty flag.

2.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the tag name.
- *get-name → String (none)*
  The *get-name* method returns the tag name.
- *clear-attribute → none (node)*
  The *clear-attribute* method clear the node attribute list.
- *add-attribute → none (Property)*
  The *add-attribute* method adds a new attribute to the tag. The attribute must be new for this method to succeed. In doubt, the *set-attribute* is preferable.
- *set-attribute → none (String Literal)*
  The *set-attribute* method sets an attribute to the tag. The first argument is the attribute name. The second argument is the attribute value. If the attribute already exists, the old value is replaced with the new one.
- *get-attribute → Property (Integer—String)*
  The *get-attribute* method returns a tag attribute in the form o a property object. With an integer object, the attribute is returned by index. With a string object, the property is return by name. If the property is not found, nil is returned.
- *get-attribute-value → String (String)*
  The *get-attribute-value* method returns a tag attribute value by name. The string argument is the attribute name. If the property is not found, an exception is raised.
- *lookup-attribute → Property (String)*
  The *lookup-attribute* method returns a tag attribute by name in the form of a property. The string argument is the attribute name. If the property is not found, an exception is raised.
- *attribute-length → Integer (none)*
  The *attribute-length* method returns the number of attributes.

3. Object XmlText

The *XmlText* class is the xml text node. A text node is directly built by the xml reader and the content placed into a string. By definition, a text node is an empty node.

3.1. Predicate.

- text-p

3.2. Inheritance.

- XmlNode

3.3. Constructors.

- *XmlText → (none)*
  The *XmlText* constructor creates a default text node. By definition, a text node is an empty node.
- *XmlText → (String)*
  The *XmlText* constructor creates a text node with the string argument.

3.4. Methods.

- *set-xval → none (String)*
  The *set-xval* method sets the text node value.
- *get-xval → String (none)*
  The *get-xval* method returns the text node value.
- *to-normal → String (none)*
  The *to-normal* method returns the normalized text node value.

4. Object XmlData

The *XmlData* class is the xml CDATA node. A data node differs from the text node in the sense that the data node contains characters that could be reserved characters such like markup delimiters. The data node is most of the time used to hold text used for scripting. The data node is an empty node.

4.1. Predicate.

- data-p

4.2. Inheritance.

- XmlNode

4.3. Constructors.

- *XmlData → (none)*
  The *XmlData* constructor creates a default data node. By definition, a data node is an empty node.
- *XmlData → (String)*
  The *XmlData* constructor creates a data node with the string argument.

4.4. Methods.

- *set-xval → none (String)*
  The *set-xval* method sets the data node value.
- *get-xval → String (none)*
  The *get-xval* method returns the data node value.

5. Object XmlComment

The *XmlComment* class is the xml comment node. The comment node is a special node that holds the comment text. The comment node is an empty node.

5.1. Predicate.

- comment-p

5.2. Inheritance.

- XmlNode

5.3. Constructors.

- *XmlComment → (none)*
  The *XmlComment* constructor creates a default comment node. By definition, a comment node is an empty node.
- *XmlComment → (String)*
  The *XmlComment* constructor creates a comment node with the string argument.

5.4. Methods.

- *set-xval → none (String)*
  The *set-xval* method sets the comment node value.
- *get-xval → String (none)*
  The *get-xval* method returns the comment node value.

6. Object XmlDoctype

The *XmlDoctype* class is the xml document type node. In its simplest form, the document type has just a name which acts the starting tag for the document. The document type can also be associated with a system or a public identifier. Note also that a local root node can be attached to this node.

6.1. Predicate.

- doctype-p

6.2. Inheritance.

- XmlNode

6.3. Constructors.

- *XmlDoctype → (String)*
  The *XmlDoctype* constructor creates a document type with a starting tag name as the string argument. This is the simplest form of a document type definition.
- *XmlDoctype → (String String)*
  The *XmlDoctype* constructor creates a document type with a starting tag name and a system identifier. The first string argument is the tag name. The second argument is the system identifier.
- *XmlDoctype → (String String String)*
  The *XmlDoctype* constructor creates a document type with a starting tag name, a public and a system identifier. The first string argument is the tag name. The second argument is the public identifier. The third argument is the system identifier.

6.4. Methods.

- *get-xval → String (none)*
  The *get-xval* method returns the document type starting tag name.
- *get-public-id → String (none)*
  The *get-public-id* method returns the document type public identifier.
- *get-system-id → String (none)*
  The *get-system-id* method returns the document type system identifier.

7. Object XmlPi

The *XmlPi* class is the xml processing node. The processing node is a tag node. Although a processing node is seen as tag with attributes, the specification describes the processing node as a special tag with a string value. The processing node is an empty node.

7.1. Predicate.

- pi-p

7.2. Inheritance.

- XmlNode

7.3. Constructors.

- *XmlPi → (String)*
  The *XmlPi* constructor creates a processing node with the name string argument.

- *XmlPi → (String String)*
  The *XmlPi* constructor creates a processing node with the name string argument and the string value. The first argument is the tag name. The second argument is the processing node value.

7.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the xml pi node name.
- *get-name → String (none)*
  The *get-name* method returns the pi node name.
- *set-xval → none (String)*
  The *set-xval* method sets the processing node value.
- *get-xval → String (none)*
  The *get-xval* method returns the processing node value.
- *map-xval → Plist (String)*
  The *map-xval* method map the processing node value to a property list.

8. Object XmlDecl

The *XmlDecl* class is the xml declaration node. The declaration node is a processing node. A declaration node is defined with a version id, an encoding string and a standalone flag. Each value is represented by an attribute at the tag level.

8.1. Predicate.

- decl-p

8.2. Inheritance.

- XmlPi

8.3. Constructors.

- *XmlDecl → (none)*
  The *XmlDecl* constructor creates a default declaration node. By default, the declaration node is set with the xml version 1.0, the UTF-8 encoding and the standalone flag is not set.
- *XmlDecl → (String)*
  The *XmlDecl* constructor creates a declaration node with a version. The string argument is the xml version version which must be a supported one.
- *XmlDecl → (String String)*
  The *XmlDecl* constructor creates a declaration node with a version and an encoding. The string argument is the xml version version which must be a supported one. The second argument is the xml encoding.
- *XmlDecl → (String String String)*
  The *XmlDecl* constructor creates a declaration node with a version, an encoding and a standalone flag. The string argument is the xml version version which must be a supported one. The second argument is the xml encoding. The third argument is the standalone flag.

9. Object XmlRef

The *XmlRef* class is the xml reference node class. This class is a base class which cannot be instantiated directly. The class is designed to hold reference, the only element which is in common is the string representation.

9.1. Predicate.

- ref-p

9.2. Inheritance.

- XmlNode

9.3. Methods.

- *set-xref → none (String)*
  The *set-xref* method sets the node reference name.
- *get-xref → String (none)*
  The *get-xref* method returns the node reference name.

10. Object XmlCref

The *XmlCref* class is the xml character reference node class. Normally this class should only be used when building a xml tree manually. During a parsing process, the character reference is automatically expanded.

10.1. Predicate.

- cref-p

10.2. Inheritance.

- XmlRef

10.3. Constructors.

- *XmlCref → (none)*
  The *XmlCref* constructor creates a default character reference those value is the null character.
- *XmlCref → (Character—Integer)*
  The *XmlCref* constructor creates a character reference those value is the character or integer argument.

10.4. Methods.

- *set-value → none (Character—Integer)*
  The *set-value* method sets the character reference value by character or integer.
- *get-value → Character (none)*
  The *get-value* method returns the character reference value.

11. Object XmlEref

The *XmlEref* class is the xml entity reference node class. The entity reference is defined with a reference name.

11.1. Predicate.

- eref-p

11.2. Inheritance.

- XmlRef

11.3. Constructors.

- *XmlEref* → *(none)*
  The *XmlCref* constructor creates an empty entity reference.
- *XmlCref* → *(String)*
  The *XmlEref* constructor creates an entity reference those value is the string argument.

12. Object XmlSection

The *XmlSection* class is the xml section type node. A section node is used to model conditional section that are part of a DTD. The section value is a string that is evaluated by the xml processor. Most of the time, it is a parameter entity reference which corresponds to the keyword INCLUDE or IGNORE , but it could be anything else. A node is also attached to this section.

12.1. Predicate.

- section-p

12.2. Inheritance.

- xmlNode

12.3. Constructors.

- *XmlSection → (String)*
  The *XmlSection* constructor creates a xml section node by value.

12.4. Methods.

- *get-xval → String (none)*
  The *get-xval* method returns the section node value.

13. Object XmlAttlist

The *XmlAttlist* class is the xml attribute list node class. A xml attribute list is primarily defined with two names. The first name is the element and the second name is the attribute name. There are 3 types of attribute list. The string type, the token type and the enumeration type. The class manages each type by associating a type descriptor which is detected at construction.

13.1. Predicate.

- attlist-p

13.2. Inheritance.

- XmlNode

13.3. Constructors.

- *XmlAttlist → (String String)*
  The *XmlAttlist* constructor creates an attribute list by element name and attribute name. The first argument is the element name. The second argument is the attribute name.

13.4. Methods.

- *set-element-name → none (String)*
  The *set-element-name* method sets the attribute list element name.
- *get-element-name → String (none)*
  The *get-element-name* method returns the attribute list element name.
- *set-attribute-name → none (String)*
  The *set-attribute-name* method sets the attribute list name.
- *get-attribute-name → String (none)*
  The *get-attribute-name* method returns the attribute list name.
- *set-type → none (String — Vector Boolean)*
  The *set-type* method set the attribute type by string or enumeration vector. In its first form, the attribute type is defined by a string. The type can be either, "CDATA", "ID", "IDREF", "IDREFS", "ENTITY", "ENTITIES", "NMTOKEN" or "NMTOKENS". In the second form, the attribute type is an enumeration those values are defined in the argument vector. The boolean argument controls the notation flag for that enumeration.
- *set-default → none (String)*
  The *set-default* method set the attribute value by string. The string can be any value or the special value "#IMPLIED" and "#REQUIRED". If the default value is fixed, the *set-fixed* is the preferred method.
- *set-fixed → none (String)*
  The *set-fixed* method set the fixed attribute default value by string.

14. Object XmlRoot

The *XmlRoot* class is the top level root instantiated by the xml reader when starting to parse a stream. There should be only one root node in a tree. The root node does not have a parent node.

14.1. Predicate.

- root-p

14.2. Inheritance.

- XmlNode

14.3. Constructors.

- *XmlRoot → (none)*
  The *XmlRoot* constructor creates a default xml root node which is empty.

14.4. Methods.

- *dup-body → XmlBody (none)*
  The *dup-body* method duplicates the root node by duplicating the root body without the declaration node.
- *declaration-p → Boolean (none)*
  The *declaration-p* predicate returns true if a declaration node exists in the root node.
- *get-declaration → XmlDecl (none)*
  The *get-declaration* method returns the declaration node associated with the root node. Normally, the declaration node is the first child node. If the declaration node does not exist, an exception is raised.
- *get-encoding → String (none)*
  The *get-encoding* method returns the root encoding mode. The root encoding mode is extracted from the declaration node, if such node exists, or the default xml system encoding is returned.

15. Object XmlDocument

The *XmlDocument* class is the xml document class. The document class is the root document class that maintains a xml document along with its associated tree and other useful information. Generally the class is constructed with a file name or a name and an input stream that is used for parsing the input data. The document can also be designed by constructing manually the document tree. In that case, the document name must be set explicitly.

15.1. Predicate.

- document-p

15.2. Inheritance.

- Nameable

15.3. Constructors.

- *XmlDocument → (none)*
  The *XmlDocument* constructor creates a default xml document.
- *XmlDocument → (String)*
  The *XmlDocument* constructor creates a xml document by parsing the file. The file name is the string argument.
- *XmlDocument → (String InputStream)*
  The *XmlDocument* constructor creates a xml document by name and by parsing the input stream. The first argument is the xml document name. The second argument is the input stream to parse.
- *XmlDocument → (String XmlRoot)*
  The *XmlDocument* constructor creates a xml document by name and root node. The first argument is the xml document name. The second argument is the xml root node.

15.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the xml document name. The *get-name* method is available from the *nameable* base class.
- *get-root → XmlRoot (none)*
  The *get-root* method returns the document xml root node.
- *get-body → XmlRoot (none)*
  The *get-body* method returns the document xml root node body without the declaration node.

16. Object XmlElement

The *XmlElement* class is the xml element class node. A xml element is represented with a name and a value. It is during the processing phase that the element value is interpreted. An element is built with a name and a value.

16.1. Predicate.

- element-p

16.2. Inheritance.

- XmlNode

16.3. Constructors.

- *XmlElement → (String String)*
  The *XmlElement* constructor creates a xml element by name and value. The first argument is the element name. The second argument is the argument value.

16.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the xml element name.
- *get-name → String (none)*
  The *get-name* method returns the element name.
- *set-xval → none (String)*
  The *set-xval* method sets the xml element value.
- *get-xval → String (none)*
  The *get-xval* method returns the element value.

17. Object XmlEntity

The *XmlEntity* class is the base class for the xml entity representation. A xml entity can be either a general entity or a parameter entity. They differ initially with the presence of the '%' character. Both entity model have a name which is path of the base class.

17.1. Predicate.

- entity-p

17.2. Inheritance.

- XmlNode

17.3. Methods.

- *set-name → none (String)*
  The *set-name* method sets the entity name.
- *get-name → String (none)*
  The *get-name* method returns the entity name.

18. Object XmlGe

The *XmlGe* class is the xml general entity node. In its simplest form, the general entity has a name and a value. The entity type can also be associated with a system or a public identifier with or without an extra type name.

18.1. Predicate.

- ge-p

18.2. Inheritance.

- XmlEntity

18.3. Constructors.

- *XmlGe → (String String)*
  The *XmlGe* constructor creates a xml entity by name and value. The first argument is the entity name. The second argument is the entity value. Most of the time, the entity value is a parameter entity.
- *XmlGe → (String String String)*
  The *XmlGe* constructor creates a xml entity by name and identifier. The first argument is the entity name. The second argument is the entity public identifier. The third argument is the entity system identifier.
- *XmlGe → (String String String String)*
  The *XmlGe* constructor creates a xml entity by name, identifier and data type. The first argument is the entity name. The second argument is the entity public identifier. the third argument is the entity system identifier. The fourth argument is the entity type name.

18.4. Methods.

- *get-xval → String (none)*
  The *get-xval* method returns the entity value.
- *get-data → String (none)*
  The *get-data* method returns the entity data type.
- *get-public-id → String (none)*
  The *get-public-id* method returns the entity public identifier.
- *get-system-id → String (none)*
  The *get-system-id* method returns the entity system identifier.

19. Object XmlPe

The *XmlPe* class is the xml parameter entity node. In its simplest form, the parameter entity has a name and a value. The entity type can also be associated with a system or a public identifier.

19.1. Predicate.

- ge-p

19.2. Inheritance.

- XmlEntity

19.3. Constructors.

- *XmlPe → (String String)*
  The *XmlGe* constructor creates a xml entity by name and value. The first argument is the entity name. The second argument is the entity value.
- *XmlPe → (String String String)*
  The *XmlGe* constructor creates a xml entity by name and identifier. The first argument is the entity name. The second argument is the entity public identifier. The third argument is the entity system identifier.

19.4. Methods.

- *get-xval → String (none)*
  The *get-xval* method returns the entity value.
- *get-public-id → String (none)*
  The *get-public-id* method returns the entity public identifier.
- *get-system-id → String (none)*
  The *get-system-id* method returns the entity system identifier.

20. Object XmlReader

The *XmlReader* class is the xml parser that operates on an input stream. The reader creates a tree of nodes by reading the input stream and returns the root node when an end-of-stream is reached. Multiple read can be done sequentially. If the reset method is not called between multiple read passes, the reader will accumulate the nodes in the current tree.

20.1. Predicate.

- reader-p

20.2. Inheritance.

- Object

20.3. Constructors.

- *XmlReader → (none)*
  The *XmlReader* constructor creates a default xml reader.

20.4. Methods.

- *reset → none (none)*
  The *reset* method resets the xml reader. In particular, the root node is restored to the nil node.
- *parse → none (InputStream—String)*
  The *parse* method parses an input stream or a file. During the parsing process, the root node is filled with the parsed nodes.
- *get-root → XmlRoot (none)*
  The *get-root* method returns the parsed root node.
- *get-node → XmlNode (String)*
  The *get-node* method parse a string and returns a node.

21. Object Xne

The *Xne* is a nameset that binds constants used by the xne system.

21.1. Constants.

- *ID → ()*
  The *ID* constant defines a node access by id.
- *PI → ()*
  The *PI* constant defines an access selector for a processing instruction node.
- *GE → ()*
  The *GE* constant defines an access selector for a general entity node.
- *TAG → ()*
  The *TAG* constant defines an access selector for a tag node.
- *ENT → ()*
  The *ENT* constant defines an access selector for an entity node.
- *EREF → ()*
  The *EREF* constant defines an access selector for an entity reference node.
- *CREF → ()*
  The *CREF* constant defines an access selector for an character reference node.
- *ELEM → ()*
  The *ELEM* constant defines an access selector for an element node.
- *TEXT → ()*
  The *TEXT* constant defines an access selector for a text node.
- *NAME → ()*
  The *NAME* constant defines a node access by name.
- *CDATA → ()*
  The *CDATA* constant defines an access selector for a character data node.
- *INDEX → ()*
  The *INDEX* constant defines a node access by child index. The child index is node index seen from the parent.

22. Object XneTree

The *XneTree* is the xne node tree manipulation class. The class operates with a node and provides numerous methods to manipulate the tree as well as changing it. Before a tree is manipulated, it is recommended to make a copy of such tree with the help of the node *copy* method.

22.1. Predicate.

- xne-tree-p

22.2. Inheritance.

- Object

22.3. Constructors.

- *XmlTree → (none)*
  The *XmlTree* constructor creates a default tree without a node.
- *XmlTree → (XmlNode)*
  The *XmlTree* constructor creates a tree with a xml node. The node stored in the object is the root of the tree subject to the operations.

22.4. Methods.

- *depth → Integer (none)*
  The *depth* method returns the depth of the tree.
- *generate-id → Integer (none)*
  The *generate-id* method generate a unique id for all node in the tree. The id attribute is set by this method.
- *set-node → none (XmlNode)*
  The *set-node* method sets the root tree node.
- *get-node → XmlNode (none)*
  The *get-node* method returns the root tree node.
- *set-attribute → none (none—String)*
  The *set-attribute* method sets an attribute to the whole tree. In the first form, the attribute is set to the whole tree. In the second form with a string argument, the attribute is set only on the tag node those name matches the name argument.
- *clear-attribute → none (none—String)*
  The *clear-attribute* method clear all attributes of the nodes in the tree. In the first form, the node attributes are cleared for all nodes in the tree. In the second form with a string argument, the attributes are cleared only with the tag node those name matches the name argument.
- *set-prefix → none (String)*
  The *set-prefix* method sets a prefix on all nodes in the tree.
- *clear-prefix → none (none)*
  The *clear-prefix* method clear the prefix for all nodes in the tree.
- *select → Vector (XneCond [Boolean])*
  The *select* method selects the node in the tree that matches the condition argument. In the first form, with one argument, the whole tree is searched. In the second form, with a boolean argument, the whole tree is searched if the second argument is false. If the boolean argument is true, the method call behaves like a call with the condition only.

23. Object XneChild

The *XneCond* is the xne condition class. The sole purpose of this class is to define one or several condition that a node must satisfy in order to be selected. The condition are accumulated in a list and later checked for a particular node. Note that an empty condition always succeeds.

23.1. Predicate.

- xne-cond-p

23.2. Inheritance.

- Object

23.3. Constructors.

- *XneCond → (none)*
  The *XneCond* constructor creates a default condition. The default condition is empty. The empty condition always succeeds.

23.4. Methods.

- *add → none (Xne [String—Integer])*
  The *add* adds a condition by type. The first argument is the condition type. The second argument is a condition information such like a string or an integer.
- *valid-p → Boolean (XmlNode)*
  The *valid-p* predicate checks that a node matches a condition. If the condition succeeds, the predicate returns true.

24. Object XsmNode

The *XsmNode* is a base class which is part of the xml simple model (xsm). In this model, a xml (or sgml, or html) text is represented by a node which can be either a tag, a text or a reference node. There is no concept of tree. The node content is stored in the form of a text string. This simple model is designed to parse weak data representation such like html text and later process it at the user discretion. The default representation is an empty text node.

24.1. Predicate.

- xsm-node-p

24.2. Inheritance.

- Object

24.3. Constants.

- *TXT → ()*
  The *TXT* constant defines a xsm text node.
- *TAG → ()*
  The *TAG* constant defines a xsm tag node.
- *REF → ()*
  The *REF* constant defines a xsm reference node.
- *END → ()*
  The *END* constant defines a xsm end node.

24.4. Constructors.

- *XsmNode → (none)*
  The *XsmNode* constructor creates a default xsm node which is an empty text node.

- *XsmNode → (String)*
  The *XsmNode* constructor creates a xsm text node by value. The string argument is the text node value
- *XsmNode → (Item String)*
  The *XsmNode* constructor creates a xsm text node by type and value. The first argument is the node type. The second argument is the node text value.

24.5. Methods.

- *text-p → Boolean (none)*
  The *text-p* predicate returns true if the node is a text node.
- *tag-p → Boolean (none)*
  The *tag-p* predicate returns true if the node is a tag node.
- *ref-p → Boolean (none)*
  The *reference-p* predicate returns true if the node is a reference node.
- *end-p → Boolean (none)*
  The *end-p* predicate returns true if the node is a reference node.
- *normal-p → Boolean (none)*
  The *normal-p* predicate returns true if the node is a normal tag node.
- *reserved-p → Boolean (none)*
  The *reserved-p* predicate returns true if the node is a reserved tag node.
- *textable-p → Boolean (none)*
  The *textable-p* predicate returns true if the node is a textable node, that is a text node or a reference node.

- *get-source-line → Integer (none)*
  The *get-source-line* method returns the node source line number if any.
- *set-source-line → none (Integer)*
  The *set-source-line* method sets the node source line number.
- *get-source-name → String (none)*
  The *get-source-name* method returns the node source name if any.
- *set-source-name → none (String)*
  The *set-source-name* method sets the node source name.
- *get-name → String (none)*
  The *get-name* method returns the next available name. name.

25. Object XsmReader

The *XmlReader* class is the simple model node reader. The reader operates with the *parse* method and returns a node or nil if the end of stream is reached. Unlike the xml reader, this reader does not build a tree and the node content is not even parsed. In this model, the node content is to be interpreted at the user discretion.

25.1. Predicate.

- xsm-reader-p

25.2. Inheritance.

- Object

25.3. Constructors.

- *XsmReader → (none)*
  The *XsmReader* constructor creates a default xsm reader. The reader is not bound to any stream.
- *XsmReader → (InputStream)*
  The *XsmReader* constructor creates a xsm reader with an input stream. The argument is the input bound to the reader.
- *XsmReader → (String)*
  The *XsmReader* constructor creates a xsm reader with an input string stream. The argument is a string which is used to create an input string stream bound to the reader.

25.4. Methods.

- *set-input-stream → none (InputStream)*
  The *set-input-stream* method bind a new input stream to the reader. Subsequent read will use the newly bound stream
- *get-node → XsmNode (none)*
  The *get-node* method parses the input stream and returns the available node.

26. Object XsmDocument

The *XsmDocument* class is the document class that maintains a xsm document along with its associated list of nodes and other useful information. Generally the class is constructed with a file name or a name and an input stream that is used for parsing the input data. When the input stream has been parsed, the nodes are stored in a vector which can be access by index.

26.1. Predicate.

- document-p

26.2. Inheritance.

- Nameable

26.3. Constructors.

- *XsmDocument → (none)*
  The *XsmDocument* constructor creates a default xsm document.
- *XsmDocument → (String)*
  The *XsmDocument* constructor creates a xsm document by name. The string argument is the file name to parse.
- *XsmDocument → (String InputStream)*
  The *XsmDocument* constructor creates a xsm document by name and by parsing the input stream. The first argument is the xsm document name. The second argument is the input stream to parse.

26.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the xsm document name. The *get-name* method is available from the *nameable* base class.
- *length → Integer (none)*
  The *length* method returns the xsm document length. The document length is the number of nodes parsed and stored in the document.
- *get-node → XsmNode (Integer)*
  The *get-node* method returns a document node by index.
- *get-info → XsoInfo (Integer [Boolean])*
  The *get-info* method returns a node info object by index. The info object is evaluated dynamically from the document node. In the first form, the node name is used to find the node end tag in order to construct the info text value. In the second form, the boolean argument, if true, forces the node name to be converted to lower case prior any comparison.
- *get-info-vector → XsoInfo (String [Boolean])*
  The *get-info-vetcor* method returns an info object vector by name. Each info object have their name that matches the string argument. The info object is evaluated dynamically from the document node. In the first form, the node name is used to match a tag node and then find the node end tag in order to construct the info text value. In the second form, the boolean argument, if true, forces the node name to be converted to lower case prior any comparison.

27. Object XsoInfo

The *XsoInfo* class is a xml/xsm information node used to carry simple information about a tag. The node is constructed by name, with a set of attributes and eventually a text associated with the node. The text information is generally the one associated between the start tag and the end tag. In the case of complex tree, such text data might be empty.

27.1. Predicate.

- xso-info-p

27.2. Inheritance.

- Nameable

27.3. Constructors.

- *XsoInfo → (none)*
  The *XsoInfo* constructor creates a default info object.
- *XsoInfo → (String)*
  The *XsoInfo* constructor creates an info object by name. The string argument is the node info name.
- *XsoInfo → (String String)*
  The *XsoInfo* constructor creates an info object by name and text. The first argument is the node info name. The second argument is the node text information.

27.4. Methods.

- *set-name → none (String)*
  The *set-name* method sets the info object name.
- *set-attribute → none (String Literal)*
  The *set-attribute* method sets an attribute by name and value. The first argument is the attribute name. The second argument is the attribute value.
- *get-attribute-list → Plist (none)*
  The *get-attribute-list* method returns the node attribute list in the form of a property list object.
- *get-attribute-value → String (String)*
  The *get-attribute-value* method returns the attribute value by name. The string argument is the attribute name.
- *set-text → none (String)*
  The *set-text* method sets the info object text.
- *get-text → String (none)*
  The *get-text* method returns the text information.

Part 2

Services

CHAPTER 23

Standard Content Session Management Service

The *Standard Content Session Management* service is an original implementation of various objects dedicated to the management of sessions, realms, identities and more generaly with data concentration in the form of binary blobs.

1. General concepts

The *afnix-csm* provides the support for manipulating content session in an ecclectic form. There are multiple types of objects which can broadly be categorized into general data management in the form of data blobs, identity and credential management and finally session management.

1.1. Blob. The concept of blob is central in the *csm* service. A blob is a registrable part. A part is an abstract taggable object uniquely identified by a uuid. The blob adds a registration identification, which enables them to be group into domains.

CHAPTER 24

Standard Content Session Management Reference

1. Object Part

The *Part* class is a taggable object which is bound by a unique key and provide a plist interface access. The part object is the foundation of the blob object and is also used to feed a collection. The key is represented by a uuid object.

1.1. Predicate.

- part-p

1.2. Inheritance.

- Taggable

1.3. Constructors.

- *Part → (none)*
  The *Part* constructor creates an empty part
- *Part → (String)*
  The *Part* constructor creates a part by name.
- *Part → (String String)*
  The *Part* constructor creates a part by name and info strings.

1.4. Methods.

- *kid-p → Boolean (String)*
  The *kip-p* predicate returns true if the part kid can be validated.
- *property-p → Boolean (String)*
  The *property-p* predicate returns true if the property name argument is defined in the part.
- *get-kid → Uuid (none)*
  The *get-kid* method returns the part kid.
- *add → none (String Literal)*
  The *add* method adds a property to the part.
- *get-header → Plist (none)*
  The *get-header* method returns the part header which is a plist with the part name, info and uuid.
- *get-plist → Plist (none)*
  The *get-plist* method returns the part plist.
- *get-value → String (String)*
  The *get-value* method returns the a part property value by name.
- *to-lteral → Literal (String)*
  The *toliteral* method returns the a part property literal by name.

2. Object Blob

The *Blob* class is a base class that models the behavior of a registered blob through the use of a registration id. The blob is registered as soon as its registration id is set. If the registration id is unset, the object is unregistered or anonymous. The registration id can be anything as long as as it is understood by the implementation that such registration is to be interpreted somewhere else. The blob is also a part which means that it has a name, info and unique key.

2.1. Predicate.

- blob-p

2.2. Inheritance.

- Part

2.3. Constructors.

- *Blob → (none)*
  The *Blob* constructor creates an empty blob.
- *Blob → (String)*
  The *Blob* constructor creates a blob by name.
- *Blob → (String String)*
  The *Blob* constructor creates a blob by name and info strings.
- *Blob → (String String String)*
  The *Blob* constructor creates a blob by rid, name and info strings.

2.4. Methods.

- *rid-p → Boolean (none)*
  The *rid-p* predicate returns true if the blob registration id is set.
- *set-rid → none (String)*
  The *set-rid* method sets the blob rid.
- *get-rid → String (none)*
  The *get-rid* method returns the blob rid.

3. Object Bloc

The *Bloc* class is a derived class which encapsulates the functionality of a blob coupled with a plist and a table of conditionals.

3.1. Predicate.

- bloc-p

3.2. Inheritance.

- Blob

3.3. Constructors.

- *Bloc → (none)*
  The *Bloc* constructor creates an empty bloc.
- *Bloc → (String)*
  The *Bloc* constructor creates a bloc by name.
- *Bloc → (String String)*
  The *Bloc* constructor creates a bloc by name and info strings.
- *Bloc → (String String String)*
  The *Bloc* constructor creates a bloc by rid, name and info strings.

3.4. Methods.

- *add-credential → none (Credential)*
  The *add-credential* method add a credential to the bloc.
- *get-credential → Credential (String)*
  The *get-credential* method returns a credential object by name.

4. Object Carrier

The *Carrier* class is a blob used to transport an object. The object transported by the carrier must be serializable.

4.1. Predicate.

- carrier-p

4.2. Inheritance.

- Blob

4.3. Constructors.

- *Carrier → (none)*
  The *Carrier* constructor creates an empty carrier.
- *Carrier → (Object)*
  The *Carrier* constructor creates a carrier with an object.
- *Carrier → (Object String)*
  The *Carrier* constructor creates a carrier with an object by name.
- *Carrier → (Object String String)*
  The *Carrier* constructor creates a carrier with an object by name and info strings.

- *Carrier → (Carrier String String String)*
  The *Carrier* constructor creates a carrier with an object by rid, name and info strings.

4.4. Methods.

- *get-object → Object (none)*
  The *get-object* method returns the carrier object.

5. Object Delegate

The *Delegate* class is a carrier blob which delegates its transport to another object. Such approach is used when the carried object needs to remains locally (aka it cannot be serialized) but a reference to it can be sent to the remote peer.

5.1. Predicate.

- delegate-p

5.2. Inheritance.

- Carrier

5.3. Constructors.

- *Delegate → (none)*
  The *Delegate* constructor creates an empty delegate.
- *Delegate → (Object)*
  The *Delegate* constructor creates a delegate with an object.
- *Delegate → (Object String)*
  The *Delegate* constructor creates a delegate with an object by name.
- *Delegate → (Object String String)*
  The *Delegate* constructor creates a delegate with an object by name and info strings.
- *Delegate → (Delegate String String String)*
  The *Delegate* constructor creates a delegate with an object by rid, name and info strings.
- *Delegate → (Delegate String String String String)*
  The *Delegate* constructor creates a delegate with an object by rid, name, info strings and delegation address.

5.4. Methods.

- *set-address → none (String)*
  The *set-address* method sets the delegate address.
- *get-address → String (none)*
  The *get-address* method returns the delegate address.

6. Object Realm

The *Realm* class is an abstract class design for the storage and management of authorities. The class provides the basic methods to create, check and validate an authority.

6.1. Predicate.

- realm-p

6.2. Inheritance.

- Nameable

6.3. Methods.

- *exists-p → Boolean (String)*
  The *exists-p* predicate checks if an authority exists by kid.
- *valid-p → Boolean (String Credential)*
  The *valid-p* predicate validates an authority by name and credential.
- *get-info → String (none)*
  The *get-info-p* method the real information string. Note that the *get-name* method is also available through the *Nameable* interface.
- *create → none (String Credential)*
  The *create* method creates an authority by name and credential.
- *update → none (Authority)*
  The *update* method updates a workzone by authority.

7. Object Session

The *Session* class is a class that defines a session to be associated with a transaction. The session object is designed to be persistent so that its data information can be retreived at any time. A session object has also the particularity to have a limited lifetime. A session object is created by name with an identifier. The session object is designed to hold a variety of parameters that are suitable for both the authentication and the session lifetime. A session is primarily defined by name with an optional information string. The session is generally associated an authentication visa which contains the session identity. The visa provides a secure mechanism compatible with a single sign on session. A session key is automatically generated when the session is created. Such key is used to generate a session hash id which can be used as a cookie value. The cookie name is also stored in the session object. When a cookie is generated, the session hash name is combined with the session hash id for the cookie production.

7.1. Predicate.

- session-p

7.2. Inheritance.

- Taggable

7.3. Constructors.

- *Session → (String)*
  The *Session* constructor creates a session by name. The string argument is the session name.
- *Session → (String String)*
  The *Session* constructor creates a session with a name and a user. The first argument is the session name. The second argument is the session information..
- *Session → (String String Integer)*
  The *Session* constructor creates a session with a name, a user and a maximum age. The first argument is the session name. The second argument is the session information. The third argument is the session maximum age expressed in seconds.

7.4. Methods.

- *expire-p → Boolean (none)*
  The *expire-p* predicate returns true if the session has expired.
- *set-hash-id → none (String)*
  The *set-hash-id* method sets the session hash identifier. The session hash id must be unique and secured enough so that the session name cannot be derived from it.
- *get-hash-id → String (none)*
  The *get-hash-id* method returns the session hash identifier.
- *set-path → none (String)*
  The *set-path* method sets the session path.
- *get-path → String (none)*
  The *get-path* method returns the session path.
- *get-max-age → Integer (none)*
  The *get-max-age* method returns the session maximum age.
- *set-max-age → none (Integer)*
  The *set-max-age* method sets the session maximum age. The maximum age is an integer in seconds relative to the current time. If the maximum age is set to 0, the session is closed.

- *get-remaining-time → Integer (none)*
  The *get-remaining-time* method returns the remaining valid session time.
- *get-expire-time → Integer (none)*
  The *get-expire-time* method returns the session expiration time in seconds. The expiration time is an absolute time.
- *set-expire-time → none (Integer)*
  The *set-expire-time* method sets the session expiration time. The expiration time is an absolute time in seconds.
- *get-creation-time → Integer (none)*
  The *get-creation-time* method returns the session creation time. The creation time is an absolute time in seconds.
- *get-modification-time → Integer (none)*
  The *get-modification-time* method returns the session creation time. The modification time is an absolute time in seconds.
- *get-cookie → Cookie (name)*
  The *get-cookie* method bakes a session cookie. The string argument is the cookie name those value is the session hash id value.
- *close → Cookie (name)*
  The *close* method close a session by reseting the session maximum age to 0. The method returns a cookie that can be used for closing the session on the peer side. The string argument is the cookie name those value is the session hash id value.
- *set-visa → None (Visa)*
  The *set-visa* method set the session visa.
- *get-visa → Visa (None)*
  The *get-visa* method returns the session visa.

CHAPTER 25

Web Application Extension Service

The *Web Application Extension* service is an original implementation that provides the support for low level HTTP transaction as well as high level XHTML page generation. The service combines various modules and provides access to the modern generation of web contents.

1. Page service objects

The *XhtmlRoot* class is the primary interface to generate *xhtml page* . The class is derived from the *XmlRoot* class and the *Mime* object. for this reason, creating a xhtml page is equivalent to add xml nodes to the page. The xhtml version is assumed to be *1.1* .

1.1. Page creation. The *XhtmlRoot* constructor takes a string argument which is the title page. When the root page is created, a *head* and *body* nodes are automatically created. Once created, it is possible to retrieve the head and body nodes from the root node. The head and body nodes are part of the *html* node which is automatically instantiated as a *XhtmlHtml* object. The html node can always be retrieved from the root node with the *get-child* xml method.

```
# create a new xhtml page
const page (afnix:wax:XhtmlRoot "AFNIX wax service")
# get the head node
const head (page:get-head)
# get the body node
const body (page:get-body)
```

The head and body nodes are part of the *html* node which is automatically instantiated as a *XhtmlHtml* object. The html node can always be retrieved from the root node with the *get-child* xml method. The root methods *get-head* and *get-body* are convenient methods that ease the page design by eliminating the references to the html node.

```
# create a new xhtml page
const page (afnix:wax:XhtmlRoot "AFNIX wax service")
# get the html node
const html (page:get-child "html")
# get the head node
const head (html:get-head)
# get the body node
const body (html:get-body)
```

1.2. Page header. The *XhtmlHead* class is the xml node that handles the xhtml head. The object is automatically created when calling the *XhtmlRoot* constructor. During the construction process, the head is automatically set with a title. Once created, the head can be filled with meta information and styles. The *add-meta* method is designed to add meta information, while the *add-style* add a link node with a style reference to the head.

```
# add a meta tag
head:add-meta "copyright" "2023"
# add a style path
head:add-style "/style.css"
```

The *add-meta* method adds a *XhtmlMeta* object which is a xml tag node. The first argument is the meta descriptor while the second argument is the meta content. Note that the *add-meta* method can be simulated by calling the *XhtmlMeta* constructor and then adding the node to the head node.

```
# create a meta node
const node (afnix:wax:XhtmlMeta "copyright" "2023")
# add the node to the head
head:add node
```

The *add-style* method adds a *XhtmlStyle* object which is a xml tag node. The string argument is the url style sheet path which gets automatically transformed to the form *@import(url)* . Note that the *add-style* method can be simulated by calling the *XhtmlStyle* constructor and then adding the node to the head node.

```
# create a style node
const node (afnix:wax:XhtmlStyle "/style.css")
# add the node to the head
head:add node
```

1.3. Page body. The *XhtmlBody* class is the xml node that handles the xhtml body. The object is automatically created when calling the *XhtmlRoot* constructor. Once created, the body node can be filled with any valid xhtml node. Since the node are initially xml tag node, it is always possible to create a tag by name and set the attributes and child nodes manually.

```
# create a new xhtml page
const page (afnix:wax:XhtmlRoot "AFNIX wax service")
# get the body node
const body (page:get-body)
# add a node
body:add-child node
```

1.4. Page emission. Since the *XhtmlRoot* object is a xml root node, the node can be used to write the complete hierarchy. The xml node node provides the *write* method that write a xml tree into a buffer of an output stream.

```
# create a new xhtml page
const page (afnix:wax:XhtmlRoot "AFNIX wax service")
# write to the output stream
page:write
```

Another mechanism for writing the page is to use the fact that the *XhtmlRoot* class implements also the *Mime* interface. With this in mind, the *XhtmlRoot* can be used within the *HttpReply* . This method is particularly useful when writing automated page generation, such like CGI scripts.

```
# create a new xhtml page
const page (afnix:wax:XhtmlRoot "AFNIX wax service")
# create an http reply object
const reply (afnix:wax:HttpReply)
# write the page as a mime object
reply:add-buffer page
# write the result to the output
reply:write
```

2. Page design objects

The *wax service module* is designed to provide several object that ease the task of creating a xhtml page. Such objects range from comment to table. Most of the time, the construction is simple the resulting node only need to be added to the page tree. When it comes to add text, the problem is becoming more subtle and is discussed later in this section.

2.1. Comment node. Adding a comment is done with the *XmlComment* class which take the comment string in the constructor. Once created, the comment node can be added to the tree.

```
# add a comment to the body node
body:add-child (
  afnix:xml:XmlComment "this is a comment")
```

2.2. Node style class. When the xhtml page is combined with the cascaded style sheet (CSS), the xhtml node tag often uses a *class* name to refer to a particular style. The *class style* is just a node attribute which can be set with the *add-attribute* method. However, most of the time, the library provides object which have the style as the first argument in the constructor. For example, the *XhtmlDiv* constructor take 0 or one argument. With one argument, the string argument is used as the style attribute.

```
# create a xhtml div with a class attribute
const div (afnix:wax:XhtmlDiv "nice")
# create a xhtml div and set the class manually
const div (afnix:wax:XhtmlDiv)
div:set-attribute "class" "nice"
```

2.3. Adding text paragraph. Adding text to a page is not a trivial task when it comes to deal with text style. By default, a piece of text is stored in the *XmlText* node. Using this node is easy. However, in a programming context, its use can become heavy. For this reason, all xml nodes provide the *parse* method which can be used to add a xml tree to the calling node. When it comes to add text that includes rendering tag, this method is quite handy.

```
# add a text with some piece in italic
node:parse "this is a <i>simple</i> method"
```

The *XhtmlPara* node is the preferred node for adding text to a xhtml page. The node takes optionally the style name in the constructor. A boolean flag can also be used to create an empty paragraph node.

```
# create a paragraph node with a style
const p (afnix:wax:XhtmlPara "title")
# add some text
p:parse "the paragraph text"
```

2.4. Adding reference. Adding reference or hyperlink to a page is achieved with the *XhtmlRef* class. Most of the time, the object is built with a uri and a text. when the node has been created, the node can be added to the page tree.

```
# create a hyperlink
const node (
  afnix:wax:XhtmlRef "http://www.afnix.org" "afnix")
# add the node in a paragraph
p:add-child node
```

2.5. Formatting elements. The *XhtmlDiv* and *XhtmlHr* classes are the basic formatting xhtml elements. The *XhtmlDiv* is a grouping element and the *XhtmlHr* is a simple horizontal ruler element. Both classes take 0 or one argument which is the style name.

```
# create a div element
const div (afnix:wax:XhtmlDiv "menu")
# create a ruler element
const hr (afnix:wax:XhtmlHr)
```

3. Managing table

The *wax service module* provides an extensive support of he xhtml table element. There are basically two strategies for creating a table. One is to use the html elements or the other is to use a print table object and automatically feed the xhtml table. The first method provides a better control while the second one is easier to use.

3.1. The table element. The *XhtmlTable* class is the class that manages xhtml table. As usual, a default style name can be specified in the constructor. Eventually, a default table row and table data default style can also be specified. Such default value are used when creating a new row with the *new-row* method.

```
# create an element with a default tr and th/td style
const tbl (afnix:wax:XhtmlTable "text" "text" "text")
# get a new row with a default style
const tr (tbl:new-row)
```

In the previous example, a table is created with a default style for the table row. When a new row is created, the default style is used for that row. If there is no default style, the row is created without a style. Note that the *new-row* method takes also a style argument that overwrites the default one.

3.2. Building the table. A table is built by adding row and data element into the rows. A row is created with the *new-row* method or the object can be constructed directly and added to the node with the *add-child* method. The *XhtmlTr* class is the table row class.

```
# get a new row with a default style
trans tr (tbl:new-row)
# create a row directly
trans tr (afnix:wax:XhtmlTr "text")
```

When a row has been created, the data can be added to the row. Normally, the *new-data* method is used to create a new table data element. If a default style is defined in the table row, the table data element is built with that style. The *new-head* method can also be used to create table header element. Again, if a default table header style exists in the table row, the element is built with that style. The *XhtmlTd* class is the table data class and the *XhtmlTh* class is the table header class.

```
# get a new data element
trans td (tr:new-data)
# create new head element
trans th (tr:new-head)
```

When the table data node has been created, the *parse* method or the *add-child* method can be called to add other nodes. another method for building the table is to use the *add-table* method which uses a print table object. In such case, the table rows and data elements are automatically added in the table.

3.3. The table structure. The table can be designed directly with table rows with table headers and table data elements. Another method, which is more structured is to use the table head, table body and table footer elements. The *XhtmlThead* class is the table head element class. The *XhtmlTbody* class is the table body element class. The *XhtmlTfoot* class is the table footer element class. These classes behaves exactly like the *XhtmlTable* and are in fact all derived from the *XhtmlTelem* class.

```
# create a xhtml table
const table (afnix:wax:XhtmlTable "text")
# create a table body
const tbody (
  afnix:wax:XhtmlTable "text" "text" "text")
# add a print tbl in the body
tbody:add-table ptbl
# add the body to the table
table:ad-child tbody
```

A table caption node can also be set with the *set-caption* method. The method simply creates a *XhtmlCaption* node and adds it to the table. The caption text is part of the method call which is used by the caption node constructor. It is also possible to create the caption node by calling the *XhtmlCaption* constructor and adding it to the table with he *add-child* method.

```
# create a xhtml table
const table (afnix:wax:XhtmlTable "text")
# set a table caption
table:set-caption "the afnix table system"
```

The table structure can also be defined with the *XhtmlCgr* class which corresponds to the xhtml column group element. The column group element is designed to support the *col* element that formats the table column.

```
# create a table
const table (afnix:wax:XhtmlTable "text")
# set the table with to 100%
table:add-attribute "width" "100%"
# create a column group
table:add-child (const xcgr (afnix:wax:XhtmlCgr))
# add a column with 30% width
cgr:add-child (afnix:wax:XhtmlCol "30%")
# add a column with 70% width
cgr:add-child (afnix:wax:XhtmlCol "70%")
```

CHAPTER 26

Web Application Extension Service Reference

1. Object XhtmlRoot

The *XhtmlRoot* class is a xml root node used for the design of a xhtml document page. At construction, the root node is initialized with a default xml processing instruction, and xhmtl node with head and body The head and body nodes can be used to add more nodes in order to build the document. The construction argument is the page title.

1.1. Predicate.

- xhtml-root-p

1.2. Inheritance.

- XmlRootMime

1.3. Constructors.

- *XhtmlRoot → (String)*
  The *XhtmlRoot* constructor creates a default xhtml page with a head and a body. The head node is set with the string title argument.

1.4. Methods.

- *get-head → XhtmlHead (none)*
  The *get-head* method returns the xhtml head node.
- *get-body → XhtmlBody (none)*
  The *get-body* method returns the xhtml body node.

2. Object XhtmlHtml

The *XhtmlHtml* class is a xhtml html node used for the design of a xhtml document page. At construction, the html node is initialized with a head node and a body node. Because a valid xhtml document must contain a title the constructor takes at least a title argument.

2.1. Predicate.

- xhtml-html-p

2.2. Inheritance.

- XmlTag

2.3. Constructors.

- *XhtmlHtml → (String)*
  The *XhtmlHtml* constructor creates a default xhtml html node with a head and a body. The head node is set with the string title argument.

2.4. Methods.

- *get-head → XhtmlHead (none)*
  The *get-head* method returns the xhtml head node.
- *get-body → XhtmlBody (none)*
  The *get-body* method returns the xhtml body node.

3. Object XhtmlHead

The *XhtmlHead* class is a xhtml head node used for the design of a xhtml document page. At construction, the head node is initialized with a with a title node. The class is designed to hold as well meta nodes and style nodes.

3.1. Predicate.

- xhtml-head-p

3.2. Inheritance.

- XmlTag

3.3. Constructors.

- *XhtmlHead → (String)*
  The *XhtmlHead* constructor creates a default xhtml head node with a title. The string argument is the head title.

3.4. Methods.

- *add-meta → none (String String)*
  The *add-meta* method adds a *XhtmlMeta* node to the head node. The first argument is the meta descriptor. The second argument is the meta contents.
- *add-style → none (String)*
  The *add-style* method adds a *XhtmlLink* node to the head node. The string argument is the style url path. The link node is automatically configured to reference a 'text/css' mime type.

4. Object XhtmlBody

The *XhtmlBody* class is a xhtml body node used for the design of a xhtml document page. The class is designed to be filled with other xhtml nodes.

4.1. Predicate.

- xhtml-body-p

4.2. Inheritance.

- XmlTag

4.3. Constructors.

- *XhtmlBody* → *(none)*
  The *XhtmlBody* constructor creates a default xhtml body node.

5. Object XhtmlTitle

The *XhtmlTitle* class is a xhtml title node used in the head node.

5.1. Predicate.

- xhtml-title-p

5.2. Inheritance.

- XmlTag

5.3. Constructors.

- *XhtmlTitle → (String)*
  The *XhtmlTitle* constructor creates a xhtml title node. The string argument is the title value. The title node is designed for the *XhtmlHead* class.

5.4. Methods.

- *set-title → none (String)*
  The *set-title* method set the node title by value.

6. Object XhtmlMeta

The *XhtmlMeta* class is a xhtml meta node used in the head node. The meta data node is an empty node with two attributes which are the descriptor and content value. The meta data is stored internally as a xml attribute.

6.1. Predicate.

- xhtml-meta-p

6.2. Inheritance.

- XmlTag

6.3. Constructors.

- *XhtmlMeta → (String String)*
  The *XhtmlMeta* constructor creates a xhtml meta node with a descriptor name and content value. The first argument is he descriptor name which is used as the node attribute name. The second argument is the content vale which is the attribute value.

7. Object XhtmlLink

The *XhtmlLink* class is a xhtml link node used in the head node. The link node is an empty node with several attributes. The most important one is the 'href' attribute that specifies the link uri. Other attributes like 'type' or 'rel' can also be set at construction.

7.1. Predicate.

- xhtml-link-p

7.2. Inheritance.

- XmlTag

7.3. Constructors.

- *XhtmlLink → (String)*
  The *XhtmlLink* constructor creates a xhtml link node by reference. The first argument is the link reference.
- *XhtmlLink → (String String)*
  The *XhtmlLink* constructor creates a xhtml link node by reference and type. The first argument is the link reference. The second argument is the link type. The link type is defined as a mime type.
- *XhtmlLink → (String String String)*
  The *XhtmlLink* constructor creates a xhtml link node by reference, type and relation. The first argument is the link reference. The second argument is the link type. The link type is defined as a mime type. The third argument is the link relation.

8. Object XhtmlStyle

The *XhtmlStyle* class is a xhtml style node used in the head node. The style node is built with a xml text node that holds the formatted url string.

8.1. Predicate.

- xhtml-style-p

8.2. Inheritance.

- XmlTag

8.3. Constructors.

- *XhtmlStyle* → *(String)*
  The *XhtmlStyle* constructor creates a xhtml style node with a url path. The string argument is the url path of the style sheet file.

9. Object XhtmlScript

The *XhtmlScript* class is a xhtml script node used in the head and body node. The script node is built with a xml tag node that holds the script content. Sometimes it is recommended to place the script inside a CDATA node that is stored as a child node of the script node. A boolean flag controls this feature at construction.

9.1. Predicate.

- xhtml-script-p

9.2. Inheritance.

- XmlTag

9.3. Constructors.

- *XhtmlScript → (String)*
  The *XhtmlScript* constructor creates a xhtml script node with a type. The string argument is the mime type string such like 'text/javascript'.
- *XhtmlScript → (String Boolean)*
  The *XhtmlScript* constructor creates a xhtml script node with a type and a CDATA node control flag. The first argument is the mime type string such like 'text/-javascript'. The second argument is the CDATA node control flag. If the flag is true, all scripts attached to the node are placed into a 'CDATA' node.
- *XhtmlScript → (String String)*
  The *XhtmlScript* constructor creates a xhtml script node with a type and a url. The first argument is the mime type string such like 'text/javascript'. The second argument is the script source url.

10. Object XhtmlPara

The *XhtmlPara* class is a xhtml paragraph node used in the body element of a xhtml page. The paragraph node can be created with a style name or as an empty node.

10.1. Predicate.

- xhtml-para-p

10.2. Inheritance.

- XmlTag

10.3. Constructors.

- *XhtmlPara → (none)*
  The *XhtmlPara* constructor creates a default xhtml paragraph node.
- *XhtmlPara → (String)*
  The *XhtmlPara* constructor creates a xhtml paragraph node with a style. The string argument is the style name.
- *XhtmlPara → (Boolean)*
  The *XhtmlPara* constructor creates an empty xhtml paragraph if the boolean argument is true.

11. Object XhtmlEmph

The *XhtmlEmph* class is a xhtml emphasize node used in the body element of a xhtml page. The emphasize node can be created with a style name.

11.1. Predicate.

- xhtml-emph-p

11.2. Inheritance.

- XmlTag

11.3. Constructors.

- *XhtmlEmph* → *(none)*
  The *XhtmlEmph* constructor creates a default xhtml emphasize node.
- *XhtmlEmph* → *(String)*
  The *Xhtmlemph* constructor creates a xhtml emphasize node with a style. The string argument is the style name.

12. Object XhtmlRef

The *XhtmlRef* class is a xhtml reference node used in the body element of a xhtml page. The node can be used to create hyperlink that references object by a uri.

12.1. Predicate.

- xhtml-ref-p

12.2. Inheritance.

- XmlTag

12.3. Constructors.

- *XhtmlRef* → *(none)*
  The *XhtmlRef* constructor creates a default xhtml reference node.
- *XhtmlRef* → *(String)*
  The *XhtmlRef* constructor creates a xhtml reference node with a uri. The string argument is the uri to use.
- *XhtmlRef* → *(String String)*
  The *XhtmlRef* constructor creates a xhtml reference node with a uri and a reference text. The first argument is the uri. The second argument is the reference text.

13. Object XhtmlImg

The *XhtmlImg* class is a xhtml image node used in the html body. The image node is an empty node with several attributes including the image source, the image width and height and an alternate string.

13.1. Predicate.

- xhtml-img-p

13.2. Inheritance.

- XmlTag

13.3. Constructors.

- *XhtmlImg → (String String)*
  The *XhtmlImg* constructor creates a xhtml image node by source and alternate name. The first argument is the image uri. The second argument is the alternate name.

13.4. Methods.

- *set-width → none (String)*
  The *set-width* method set the image width attribute.
- *set-height → none (String)*
  The *set-height* method set the image height attribute.
- *set-geometry → none (String)*
  The *set-geometry* method set the image width and height attribute in one call.

14. Object XhtmlDiv

The *XhtmlDiv* class is a xhtml *div* node used in the body element of a xhtml page. The *div* node is a xhtml grouping element.

14.1. Predicate.

- xhtml-div-p

14.2. Inheritance.

- XmlTag

14.3. Constructors.

- *XhtmlDiv* → *(none)*
  The *XhtmlDiv* constructor creates a default xhtml div node.
- *XhtmlDiv* → *(String)*
  The *XhtmlDiv* constructor creates a xhtml div node with a style. The string argument is the style name.

15. Object XhtmlPre

The *XhtmlPre* class is a xhtml *pre* node used in the body element of a xhtml page. The *pre* node is a xhtml formatting element.

15.1. Predicate.

- xhtml-pre-p

15.2. Inheritance.

- XmlTag

15.3. Constructors.

- *XhtmlPre* → *(none)*
  The *XhtmlPre* constructor creates a default xhtml pre node.
- *XhtmlPre* → *(String)*
  The *XhtmlPre* constructor creates a xhtml pre node with a style. The string argument is the style name.

16. Object XhtmlHr

The *XhtmlHr* class is a xhtml *hr* node used in the body element of a xhtml page. The *hr* node is a xhtml horizontal ruler element.

16.1. Predicate.

- xhtml-hr-p

16.2. Inheritance.

- XmlTag

16.3. Constructors.

- *XhtmlHr* → *(none)*
  The *XhtmlHr* constructor creates a default xhtml hr node.
- *XhtmlHr* → *(String)*
  The *XhtmlHr* constructor creates a xhtml hr node with a style. The string argument is the style name.

17. Object XhtmlCgr

The *XhtmlCgr* class is a xhtml column group node used in the table element. The column group is designed to hold the column definition bound by the *XhtmlCol* class.

17.1. Predicate.

- xhtml-cgr-p

17.2. Inheritance.

- XmlTag

17.3. Constructors.

- *XhtmlCgr* → *(none)*
  The *XhtmlCgr* constructor creates a default xhtml *colgroup* node.

18. Object XhtmlCol

The *XhtmlCol* class is a xhtml column node used in the table column group element.

18.1. Predicate.

- xhtml-col-p

18.2. Inheritance.

- XmlTag

18.3. Constructors.

- *XhtmlCol* → *(none)*
  The *XhtmlCol* constructor creates a default xhtml *col* node.
- *XhtmlCol* → *(String)*
  The *XhtmlCol* constructor creates a xhtml *col* node with a string width argument. The argument is the *width* attribute value.

19. Object XhtmlTh

The *XhtmlTh* class is a xhtml *th* node used in the table row. The object can be built with a style name.

19.1. Predicate.

- xhtml-th-p

19.2. Inheritance.

- XmlTag

19.3. Constructors.

- *XhtmlTh* → *(none)*
  The *XhtmlTh* constructor creates a default xhtml th node.
- *XhtmlTh* → *(String)*
  The *XhtmlTh* constructor creates a xhtml th node with a style. The string argument is the style name.

20. Object XhtmlTd

The *XhtmlTd* class is a xhtml *td* node used in the table row. The object can be built with a style name.

20.1. Predicate.

- xhtml-td-p

20.2. Inheritance.

- XmlTag

20.3. Constructors.

- *XhtmlTd → (none)*
  The *XhtmlTd* constructor creates a default xhtml td node.
- *XhtmlTd → (String)*
  The *XhtmlTd* constructor creates a xhtml td node with a style. The string argument is the style name.

21. Object XhtmlTr

The *XhtmlTr* class is a xhtml *tr* node used in the table node. The table row node is designed to accumulate table head or table data nodes.

21.1. Predicate.

- xhtml-tr-p

21.2. Inheritance.

- XmlTag

21.3. Constructors.

- *XhtmlTr → (none)*
  The *XhtmlTr* constructor creates a default xhtml tr node.
- *XhtmlTr → (String)*
  The *XhtmlTr* constructor creates a xhtml tr node with a style. The string argument is the style name.
- *XhtmlTr → (String String)*
  The *XhtmlTr* constructor creates a xhtml tr node with a style and a default table data style. The string argument is the table row style name. The second argument is the default table data style.

21.4. Methods.

- *new-head → XhtmlTh (none — String)*
  The *new-head* method returns a new table head data object. Without argument, a default *XhtmlTh* object is created. With a string argument, the *XhtmlTh* object is constructed with a style name.
- *new-data → XhtmlTd (none — String)*
  The *new-data* method returns a new table data object. Without argument, a default *XhtmlTd* object is created. With a string argument, the *XhtmlTd* object is constructed with a style name.
- *set-head-class → none (String)*
  The *set-head-class* method sets the default table head style. The default style is use with the *new-head* method.
- *set-data-class → none (String)*
  The *set-data-class* method sets the default table data style. The default style is use with the *new-data* method.
- *set-xdef-class → none (String)*
  The *set-xdef-class* method sets the default table head and data style. The default style is use with the *new-head* and *new-data* methods. This method combines the *set-head-class* and the *set-head-class*

22. Object XhtmlTelem

The *XhtmlTelem* class is an abstract class that implements the node behavior for the table head, body, foot and table elements. The table element node is designed to accumulate table row nodes. This class cannot be constructed directly.

22.1. Predicate.

- xhtml-telem-p

22.2. Inheritance.

- XmlTag

22.3. Methods.

- *new-row → XhtmlTr (none — String)*
  The *new-row* method returns a new table row object. Without argument, a default *XhtmlTr* object is created. With a string argument, the *XhtmlTr* object is constructed with a style name.
- *add-table → none (PrintTable [Boolean])*
  The *add-table* method adds a print table into the table element by adding automatically the row and the associated formatting information such like the data direction. The optional second argument controls whether or not the table tag shall be used to build reference node for the table elements.
- *set-xrow-class → none (String)*
  The *set-xrow-class* method sets the default table row data style. The default row style is use with the *new-row* method.
- *set-xdef-class → none (String)*
  The *set-xdef-class* method sets the default table head and data style. The default style is use with the *new-row* method to set the table head and data default style.

23. Object XhtmlThead

The *XhtmlThead* class is a xhtml thead node. The table head node is designed to accumulate table rows nodes. The class acts almost like the xhtml table class.

23.1. Predicate.

- xhtml-thead-p

23.2. Inheritance.

- XhtmlTelem

23.3. Constructors.

- *XhtmlThead → (none)*
  The *XhtmlThead* constructor creates a default xhtml table head node.
- *XhtmlThead → (String)*
  The *XhtmlThead* constructor creates a xhtml table head node with a style. The string argument is the style name.
- *XhtmlThead → (String String)*
  The *XhtmlThead* constructor creates a xhtml table head node with a style and a default table row style. The string argument is the table head style name. The second argument is the default table row style.
- *XhtmlThead → (String String String)*
  The *XhtmlThead* constructor creates a xhtml table head node with a style, a default table row style and a default table data style. The string argument is the table head style name. The second argument is the default table row style. The third argument is the table data style.

24. Object XhtmlTbody

The *XhtmlTbody* class is a xhtml tbody node. The table body node is designed to accumulate table rows nodes. The class acts almost like the xhtml table class.

24.1. Predicate.

- xhtml-tbody-p

24.2. Inheritance.

- XhtmlTelem

24.3. Constructors.

- *XhtmlTbody → (none)*
  The *XhtmlTbody* constructor creates a default xhtml table body node.
- *XhtmlTbody → (String)*
  The *XhtmlTbody* constructor creates a xhtml table body node with a style. The string argument is the style name.
- *XhtmlTbody → (String String)*
  The *XhtmlTbody* constructor creates a xhtml table body node with a style and a default table row style. The string argument is the table body style name. The second argument is the default table row style.
- *XhtmlTbody → (String String String)*
  The *XhtmlTbody* constructor creates a xhtml table body node with a style, a default table row style and a default table data style. The string argument is the table body style name. The second argument is the default table row style. The third argument is the table data style.

25. Object XhtmlTfoot

The *XhtmlTfoot* class is a xhtml tfoot node. The table foot node is designed to accumulate table rows nodes. The class acts almost like the xhtml table class.

25.1. Predicate.

- xhtml-tfoot-p

25.2. Inheritance.

- XhtmlTelem

25.3. Constructors.

- *XhtmlTfoot → (none)*
  The *XhtmlTfoot* constructor creates a default xhtml table foot node.
- *XhtmlTfoot → (String)*
  The *XhtmlTfoot* constructor creates a xhtml table foot node with a style. The string argument is the style name.
- *XhtmlTfoot → (String String)*
  The *XhtmlTfoot* constructor creates a xhtml table foot node with a style and a default table row style. The string argument is the table foot style name. The second argument is the default table row style.
- *XhtmlTfoot → (String String String)*
  The *XhtmlTfoot* constructor creates a xhtml table foot node with a style, a default table row style and a default table data style. The string argument is the table foot style name. The second argument is the default table row style. The third argument is the table data style.

26. Object XhtmlTable

The *XhtmlTable* class is a xhtml table node. The table node is designed to accumulate table row nodes or column group nodes. The table can also be designed with a table head, body and foot nodes.

26.1. Predicate.

- xhtml-table-p

26.2. Inheritance.

- XhtmlTelem

26.3. Constructors.

- *XhtmlTable → (none)*
  The *XhtmlTable* constructor creates a default xhtml table foot node.
- *XhtmlTable → (String)*
  The *XhtmlTable* constructor creates a xhtml table foot node with a style. The string argument is the style name.
- *XhtmlTable → (String String)*
  The *XhtmlTable* constructor creates a xhtml table foot node with a style and a default table row style. The string argument is the table foot style name. The second argument is the default table row style.
- *XhtmlTable → (String String String)*
  The *XhtmlTable* constructor creates a xhtml table foot node with a style, a default table row style and a default table data style. The string argument is the table foot style name. The second argument is the default table row style. The third argument is the table data style.

26.4. Methods.

- *set-caption → none (String)*
  The *set-caption* method sets the table caption. A new *XhtmlCaption* node is automatically added to the table tree during this method call.

27. Object XmlMime

The *XmlMime* class is a generic xml mime document class. The class is used to construct a mime version of a xml document which can be obtained from a file name, or an input stream. By default, the mime type 'application/xml'.

27.1. Predicate.

- xml-mime-p

27.2. Inheritance.

- XmlDocumentMime

27.3. Constructors.

- *XmlMime* → *(none)*
  The *XmlMime* constructor creates a default xml mime document.
- *XmlMime* → *(String)*
  The *XmlMime* constructor creates a xml mime document by parsing a file. The file name is the string argument.
- *XmlMime* → *(String InputStream)*
  The *XmlMime* constructor creates a xml mime document by name and by parsing the input stream. The first argument is the xml document name. The second argument is the input stream to parse.

28. Object XhtmlMime

The *XhtmlMime* class is a generic xhtml mime document class. The class is used to construct a mime version of a xhtml document which can be obtained from a file name, or an input stream. By default, the mime type 'application/xhtml+xml'.

28.1. Predicate.

- xhtml-mime-p

28.2. Inheritance.

- XmlMime

28.3. Constructors.

- *XhtmlMime → (none)*
  The *XhtmlMime* constructor creates a default xhtml mime document.
- *XhtmlMime → (String)*
  The *XhtmlMime* constructor creates a xhtml mime document by parsing a file. The file name is the string argument.
- *XhtmlMime → (String InputStream)*
  The *XhtmlMime* constructor creates a xhtml mime document by name and by parsing the input stream. The first argument is the xhtml document name. The second argument is the input stream to parse.

29. Object XhtmlForm

The *XhtmlForm* class is a generic xhtml form object. A form is defined by an action and a method. When the form is created, it is appropriate to add other xhtml objects.

29.1. Predicate.

- xhtml-form-p

29.2. Inheritance.

- XhtmlTag

29.3. Constructors.

- *XhtmlForm → (String String)*
  The *XhtmlForm* constructor creates a xhtml form by action and method. The first argument is the uri path for the action while the second argument is the method to use for the action.

30. Object XhtmlText

The *XhtmlText* class is a generic xhtml input text object. An input text is a form element which is used to capture text in a field. The text value is attached with the name attribute.

30.1. Predicate.

- xhtml-text-p

30.2. Inheritance.

- XhtmlTag

30.3. Constructors.

- *XhtmlText → (String)*
  The *XhtmlText* constructor creates a xhtml input text by name.
- *XhtmlText → (String String)*
  The *XhtmlText* constructor creates a xhtml input text by name and size. The first argument is the input text name and the second argument is the text field size.

30.4. Methods.

- *set-size → none (String)*
  The *set-size* method sets the input text size.

31. Object XhtmlSubmit

The *XhtmlSubmit* class is a generic xhtml input submit object. An input submit object is a button which is used inside a form generally as a condition to activate the form.

31.1. Predicate.

- xhtml-submit-p

31.2. Inheritance.

- XhtmlTag

31.3. Constructors.

- *XhtmlSubmit → (String)*
  The *XhtmlSubmit* constructor creates a xhtml submit button by value.
- *XhtmlText → (String String)*
  The *XhtmlText* constructor creates a xhtml submit button by value and size. The first argument is the input submit value and the second argument is the submit size.

31.4. Methods.

- *set-size → none (String)*
  The *set-size* method sets the submit button size.

CHAPTER 27

XML Processing Environment Service

The *XML Processing Environment* service is an original implementation that provides the support for processing xml document that could be accessed locally or remotely. In particular, the processing environment supports the *XML include* facility.

1. XML content

The *XmlContent* class is an extension of the XML document object that provides the service for loading a XML document locally or from the Internet. The class operates with an uri, which permits to selects the appropriate loader from the uri scheme.

1.1. Content creation. The *XmlContent* operates with an uri that permits to select the appropriate loader. If the uri scheme is a file scheme, the content is retrieved locally. If the uri scheme is http, the content is retrieved by establishing a http connection over the Internet.

```
# create a document from a local file
const xdoc (
  afnix:xpe:XmlContent "file:///home/afnix/file.xml")
```

When the uri scheme is a file, the uri authority is empty (hence the double //) and the path indicates the file to parse. The XmlContent object is derived from the *XmlDocument* object which contains the parsed tree with the *XmlRoot* object.

```
# create a document from a http connection
const xdoc (
  afnix:xpe:XmlContent
  "http://www.afnix.org/index.xht")
```

When the uri scheme is a http scheme, the document is downloaded by establishing an http connection with the uri authority. When the http header is received, the content is parsed to create a valid xml document. If the http response header indicates that the page has moved and a new location is provided, the object manages automatically to follow such location.

1.2. Content and document name. Since the *XmlContent* object is derived from the *XmlContent* object, the content object is defined with a uri name and a document name. Under normal circumstances, the document name is derived from the content name by normalizing it. The content name is the object constructor name, while the document name is the normalized document name. The *get-name* method returns the content name while the *get-document-name* method returns the document name.

```
# create a document by name
const xdoc (afnix:xpe:XmlContent "file" "file.xml")
```

In the previous example, a xml content object is created by name with a document name. It is the document name that gets normalized. Therefore in the previous example, the `file.xml` document name is normalized into a file uri. The normalization rule always

favor the file scheme. This means that without a scheme, the file scheme is automatically added.

1.3. Content type. Many times, the content type cannot be detected from the uri name. Once opened, if the content header provides a clue about the content type, the opened input stream get adjusted automatically to reflect this fact. However, this situation does not occurs often and with http scheme, the content type header response does not often provides the character encoding associated with the stream. For this reason, the *XmlContent* constructor provides a mechanism to accept the encoding mode.

```
# create a new content by name and encoding mode
const xdoc (
  afnix:xpe:XmlContent "file" "file.xml" "UTF-8")
```

CHAPTER 28

XML Processing Environment Service Reference

1. Object XmlContent

The *XmlContent* class is an extension of the xml document class that operates at the uri level. If the uri is a local file the xml document is created from an input file stream. If the uri is an url, the content is fetched automatically. The class constructors permit to separate the content name from the document name and also to specify the content encoding.

1.1. Predicate.

- xml-content-p

1.2. Inheritance.

- XmlDocumentMime

1.3. Constructors.

- *XmlContent → (String)*
  The *XmlContent* constructor creates a xml document by name. The document name is the normalized uri name that always favor a file scheme in the absence of it.
- *XmlContent → (String String)*
  The *XmlContent* constructor creates a xml document by name. The first argument is the content name. The second argument is the document name which is normalized to form the uri name used to load the document.
- *XmlContent → (String String String)*
  The *XmlContent* constructor creates a xml document by name and encoding mode. The first argument is the content name. The second argument is the document name which is normalized to form the uri name used to load the document. The third argument is the content character encoding.

1.4. Methods.

- *get-document-uri → String (none)*
  The *get-document-uri* method returns the document normalized uri.
- *get-document-name → String (none)*
  The *get-document-name* method returns the object document name. This method complements the *get-name* method which returns the object name.

2. Object XmlFeature

The *XmlFeature* class is a xml processor base class that defines a processing feature. A processing feature is defined by name and information with a processing level. The default processing level is null. When the processor is called, it calls sequentially and in ascending order all features.

2.1. Predicate.

- xhtml-feature-p

2.2. Inheritance.

- Nameable

2.3. Methods.

- *get-info → String (none)*
  The *get-info* method returns the xml feature information string. The feature name is available from the *get-name* provided by the *Nameable* base class.
- *set-processing-level → none (Integer)*
  The *set-processing-level* method sets the feature processing level. The integer argument is the level to set.
- *get-processing-level → Integer (none)*
  The *get-processing-level* method returns the feature processing level.
- *processing-level-p → Boolean (Integer)*
  The *processing-level-p* predicate returns true if the integer argument equal the feature processing level.
- *process → XmlContent (XmlContent)*
  The *process* method process the input xml content and returns a new xml content. The method is automatically called by the xml processor.

3. Object XmlProcessor

The *XmlProcessor* class is a global class designed to operate on a xml content. The xml processor provides several features that can be enabled prior the document processor. Once the features are defined, the 'process' method can be called and a new xml content can be produced.

3.1. Predicate.

- xml-processor-p

3.2. Inheritance.

- Object

3.3. Constructors.

- *XmlProcessor → (none)*
  The *XmlProcessor* constructor creates a default xml processor without any feature.

3.4. Methods.

- *feature-length → Integer (none)*
  The *feature-length* method returns the number of features defined in the xml processor.
- *add-feature → none (XmlFeature)*
  The *add-feature* method adds a feature object to the processor. The feature processing level does not have to be sorted prior the insertion. Adding multiple feature creates a processor chain.
- *get-feature → XmlFeature (Integer)*
  The *get-feature* method return a processor feature by index.
- *process → XmlFeature (XmlFeature)*
  The *process* method create a new xml content by calling processing feature chain. The feature chain consists of feature object sorted in ascending order. If the processor contains only one feature, calling the *process* method is equivalent to call the *XmlFeature* same method.

4. Object XmlInclude

The *XmlInclude* class is a xml processor feature class designed to handle the "XInclude" schema that permits to include xml document. The feature operates recursively by scanning the document for a "xi:include" tag and replacing the content by the appropriate tree. The feature operates recursively unless specified otherwise.

4.1. Predicate.

- xml-include-p

4.2. Inheritance.

- XmlFeature

4.3. Constructors.

- *XmlInclude* → *(none)*
  The *XmlInclude* constructor creates a default xml include feature. The default feature processing level is 10.
- *XmlInclude* → *(Integer)*
  The *XmlInclude* constructor creates a xml include feature with a processing level. The integer argument is the feature processing level to set.

Index