

**TIPSTER Text Architecture Design**

**Version 3.1 7 October 1998**

*Ralph Grishman*

*New York University*

grishman@cs.nyu.edu

*and the  
TIPSTER Phase III Contractors*







# Table of Contents

|                                                |            |
|------------------------------------------------|------------|
| 1.0 GOALS                                      | <u>14</u>  |
| 2.0 CONCEPTS                                   | <u>33</u>  |
| 2.1 Object Classes                             | <u>33</u>  |
| 2.2 Optionality                                | <u>33</u>  |
| 2.3 Correspondence to Interface Specifications | <u>33</u>  |
| 2.3.1 Optional Arguments                       | <u>44</u>  |
| 2.3.2 Implementation of Sequences              | <u>44</u>  |
| 2.4 Storage Management                         | <u>45</u>  |
| 2.5 Error Handling                             | <u>55</u>  |
| 3.0 BASIC CLASSES                              | <u>77</u>  |
| 3.1 Attributes                                 | <u>77</u>  |
| Class Attribute                                | <u>77</u>  |
| Class AttributeValue                           | <u>77</u>  |
| Abstract Class ObjectReference                 | <u>77</u>  |
| Class CollectionReference                      | <u>77</u>  |
| Class Document Reference                       | <u>77</u>  |
| Class AttributeReference                       | <u>88</u>  |
| Class AnnotationReference                      | <u>88</u>  |
| Abstract Class AttributedObject                | <u>88</u>  |
| 3.2 Persistent Objects                         | <u>88</u>  |
| Abstract Class PersistentObject                | <u>88</u>  |
| 3.3 Byte Sequences                             | <u>99</u>  |
| Class ByteSequence                             | <u>99</u>  |
| 4.0 DOCUMENTS AND COLLECTIONS                  | <u>114</u> |
| 4.1 Documents                                  | <u>114</u> |
| Class Document                                 | <u>114</u> |
| 4.2 Collections                                | <u>124</u> |
| Class Collection                               | <u>124</u> |
| 5.0 DOCUMENT ANNOTATIONS: GENERAL STRUCTURE    | <u>154</u> |
| 5.1 What Is Annotated?                         | <u>154</u> |
| Class Span                                     | <u>154</u> |
| 5.1.1 Code Sets and Character Positions        | <u>154</u> |
| 5.1.2 Modification of the Text                 | <u>154</u> |
| 5.2 Information Associated With an Annotation  | <u>164</u> |
| Class Annotation                               | <u>164</u> |
| 5.3 Accessing Annotations                      | <u>164</u> |
| Class AnnotationSet                            | <u>164</u> |
| 5.4 Annotation Type Declarations               | <u>174</u> |
| 5.5 Examples of Annotations                    | <u>184</u> |
| 5.6 Invoking Annotators                        | <u>212</u> |
| 5.7 External Representation of Annotations     | <u>212</u> |
| 5.8 Annotation Schemata and Style Sheets       | <u>212</u> |
| 6.0 TYPES OF DOCUMENT ANNOTATIONS              | <u>232</u> |
| 6.1 Annotations for Major Document Elements    | <u>232</u> |
| 6.2 Annotations for Document Header Elements   | <u>242</u> |
| 6.3 Structural Annotations                     | <u>242</u> |
| 6.4 Sub-paragraph Annotations                  | <u>252</u> |
| 6.5 Common Document Attributes                 | <u>252</u> |
| 6.6 Common Annotation Attributes               | <u>262</u> |
| 6.7 Example of Annotations and Attributes      | <u>262</u> |
| 7.0 DETECTION                                  | <u>292</u> |
| 7.1 Object Classes                             | <u>292</u> |
| 7.1.1 Detection Needs and Queries              | <u>292</u> |

|                                                     |                 |
|-----------------------------------------------------|-----------------|
| 7.1.2 Detection Needs                               | <del>2929</del> |
| Class DetectionNeed                                 | <del>2929</del> |
| Class DetectionNeedCollection                       | <del>3030</del> |
| 7.1.3 Queries                                       | <del>3131</del> |
| Class DetectionQuery                                | <del>3131</del> |
| Class RetrievalQuery                                | <del>3131</del> |
| Class RoutingQuery                                  | <del>3132</del> |
| 7.1.4 Document and Query Indexes                    | <del>3232</del> |
| Class DocumentCollectionIndex                       | <del>3232</del> |
| Class QueryCollectionIndex                          | <del>3333</del> |
| 7.1.5 Query Monitoring                              | <del>3434</del> |
| Class Monitor                                       | <del>3434</del> |
| 7.2 Functional Model                                | <del>3435</del> |
| 7.2.1 Retrospective Retrieval                       | <del>3535</del> |
| 7.2.2 Routing                                       | <del>3535</del> |
| 7.2.3 Relevance Feedback                            | <del>3636</del> |
| 8.0 EXTRACTION                                      | <del>3939</del> |
| 8.1 Representing Templates as Annotations           | <del>3939</del> |
| 8.2 An Example                                      | <del>3939</del> |
| APPENDIX A POSSIBLE EXTENSIONS TO THE ARCHITECTURE  | <del>4141</del> |
| A.1 Enforcing Type Declarations                     | <del>4141</del> |
| A.2 Customizable Extraction Systems                 | <del>4141</del> |
| A.2.1 Object Classes                                | <del>4141</del> |
| Class ExtractionNeed                                | <del>4141</del> |
| Class CustomizedExtractionSystem                    | <del>4141</del> |
| Class Template Object Library                       | <del>4242</del> |
| A.2.2 Functional Model                              | <del>4242</del> |
| APPENDIX B CLASSES AND THEIR OPERATIONS             | <del>4343</del> |
| Class Annotation                                    | <del>4343</del> |
| Class AnnotationReference                           | <del>4343</del> |
| Class AnnotationSet                                 | <del>4343</del> |
| Class Attribute                                     | <del>4343</del> |
| Class AttributeReference                            | <del>4344</del> |
| Class AttributeValue                                | <del>4444</del> |
| Class ByteSequence                                  | <del>4444</del> |
| Class Collection                                    | <del>4444</del> |
| Class CollectionReference                           | <del>4444</del> |
| Class DetectionNeed                                 | <del>4444</del> |
| Class DetectionNeedCollection                       | <del>4444</del> |
| Class DetectionQuery                                | <del>4445</del> |
| Class Document                                      | <del>4545</del> |
| Class DocumentCollectionIndex                       | <del>4545</del> |
| Class Document Reference                            | <del>4545</del> |
| Class Monitor                                       | <del>4545</del> |
| Class QueryCollectionIndex                          | <del>4546</del> |
| Class RetrievalQuery                                | <del>4646</del> |
| Class RoutingQuery                                  | <del>4646</del> |
| Class Span                                          | <del>4646</del> |
| APPENDIX C C LANGUAGE HEADER FILE                   | <del>4747</del> |
| APPENDIX D PATTERN SPECIFICATION LANGUAGE           | <del>5353</del> |
| APPENDIX E - ALTERNATE DETECTION NEED SPECIFICATION | <del>5959</del> |

## 1.0 GOALS

The TIPSTER Program aims to push the technology for access to information in large (multi-GB) text collections, in particular for the analysts in Government agencies. Technology is being developed for document detection ("information retrieval") and for data extraction from free text.

The primary mission of the TIPSTER Common Architecture is to provide a vehicle for efficiently delivering this detection and extraction technology to the Government agencies. The Architecture also has a secondary mission of providing a convenient and efficient environment for research in document detection and data extraction.

To accomplish this mission, the TIPSTER Architecture is being designed to:

- provide APIs for document detection, data extraction, and the associated document management functions
- support monolingual and multilingual applications
- allow the interchange of modules from different suppliers ("plug and play")
- apply to a wide range of software and hardware environments
- scale to a wide range of volumes of document archives and of document flow
- support appropriate application response time
- support incorporation of multi-level security
- enhance detection and extraction through the exchange of information, and through easier access to linguistic annotations



## 2.0 CONCEPTS

The architecture is described by a set of object classes and a set of functions associated with these objects. In addition, there is a "functional" section which indicates how data typically flows between these functions.

### 2.1 Object Classes

An object class is characterized by a class name, a set of named properties, and a set of operations. Unless explicitly noted otherwise, there is an operation (the property accessor function) associated with each property for reading that property's value. If the property is followed by (R, W), operations are provided both for reading and for writing that property. If the property is followed by ( $\mathbb{R}$ ), no functions are provided for reading or writing the property.

Each property has a value, which may be

- an object (of one or several classes)
- a sequence of objects (ordered), denoted by "sequence of..."
- a string (of characters)
- an integer
- a float (real number)
- a byte
- a Boolean value (true or false)
- a member of an enumerated type, denoted by "one of { ... }"
- *nil*

The operations will include both procedures (which do not return a value) and functions (which do). The notation is

procedure (type of arg1, type of arg2, ...)

function (type of arg1, type of arg2, ...): type of result

To indicate the significance of particular arguments, an argument position may contain

argument name: argument type

If a class  $C_1$  is a subclass of another class  $C_2$  (indicated by the notation Type of  $C_2$  in the definition of  $C_1$ ) then  $C_1$  inherits all the properties and operations of  $C_2$ .

The designation of a class as an **Abstract Class** indicates that the class is not intended to be instantiated but is intended to serve as a superclass for other classes (which will be instantiated).

A class C can include operations whose name has the form "class.C". If D is a type of C (i.e., class D includes the specification Type of C), then the operation as inherited by D has the name "class.D". This facility is provided to allow for the specialization of operations which create new instances of a class.

### 2.2 Optionality

Some objects and functions will be required: they must be implemented by any system conforming to the architecture. Some objects and functions will be *optional*: they need not be included, but if they are, they must conform to the standard. This allows us to define standards, for example, for some linguistic annotations, without requiring all systems to generate such annotations.

### 2.3 Correspondence to Interface Specifications

This document provides an abstract definition of the architecture in terms of classes and operations. This architecture will be implemented in a number of programming languages; currently implementations are being developed in C, Tcl, and Common Lisp. This section describes the correspondence between the set of operations described in this document and the APIs for implementations of this architecture in these programming languages.

**Common Lisp:** The classes, properties, and operations defined herein correspond to those of a Common Lisp implementation of the TIPSTER Architecture as follows:

1. (because Lisp is normally not case sensitive) each capital letter in the name of a class, property, or operation, except for the first letter in a name, will be preceded by a "-" in Lisp
2. each class, property, and operation corresponds to a Lisp class, property, and function
3. each argument of the form "*class*" becomes a positional argument; each argument of the form "name: *class*" becomes a keyword argument with the keyword name

4. sequences are represented as lists

In Lisp, the name of the property accessor function is formed from the class name, a hyphen, and the property name (e.g., **attribute-name** and **attribute-value**). If the property is writeable, the property accessor function acts as a "generalized variable" which can be set by **setf**; e.g., (**setf (collection-owner collection1) "Mitchell"**).

**C:** Each operation defined herein corresponds to a C function, with the same name as in the abstract architecture. All arguments in the C implementation are positional; the argument names ("keywords") in the abstract architecture are not used. If property *Comp* of a class is readable, it is accessed by the function `Get Comp`; if it is also writeable, it is set by the function `Set Comp`.

Note that the abstract architecture occasionally "overloads" operations: the same operation name may apply to different classes of arguments. To support such overloading, the C implementations of the various classes, as well as sets and sequences, should employ a generic container structure which will allow a C function to determine the class of an actual argument.<sup>1</sup>

The C-language typing, including the overloading of various functions, is spelled out in Appendix C.

**Tcl:** Operation names and argument lists in Tcl shall be the same as in the C implementation.

### 2.3.1 Optional Arguments

In addition to the arguments which are specified for each operation in this document and which are required, an implementation of the Architecture may provide optional keyword or positional arguments for any of the operations. The operation must be able to complete and to perform the specified function even if only the required arguments are given, but use of the optional arguments may provide enhanced performance or a greater range of functionality.

### 2.3.2 Implementation of Sequences

The architecture includes the notion 'sequence of X', where X is a type, as one of the possible values of an argument to an operation or the value of a property. In describing an implementation of the Architecture (an API), it is necessary to specify the representation or set of operations for such sequences.

The C language interface (Appendix C) defines types *AttributeSet*, *AttributeValueSet*, *DocumentCollectionIndexSet*, *QueryCollectionIndex*, *SpanSet*, and *stringSet*, corresponding to 'sequence of Attribute', 'sequence of AttributeValue', 'sequence of DocumentCollectionIndex', 'sequence of QueryCollectionIndex', 'sequence of Span', and 'sequence of string' in the Architecture<sup>2</sup>. These are referred to collectively as *XSets*, where X may be *Attribute*, *Span*, etc. An empty *XSet* is created by the operation

`CreateXSet ()`: *XSet*

(i.e., by one of the operations `CreateAttributeSet`, `CreateSpanSet`, etc.). The following operations apply to *XSets*:

`Nth (XSet, n: integer)`: *X*

returns the nth element of *XSet* (where the first element of sequence has index 0)

`Push (XSet, X)`

adds X to the end of sequence *XSet*

`Pop (XSet)`: *X*

removes and returns the last element of *XSet*

`Length (XSet)`: integer

returns the length of *XSet*

In addition, the operation *Free*, described just below, applies to all types of objects, including *XSets*.

## 2.4 Storage Management

A *free* operation must be provided for *every* class of object to release the memory associated with that object as well as to perform any necessary implementation specific cleanup operations.

---

<sup>1</sup> The overloading does not extend to basic C data types (char, int, float), since these could not be differentiated from structures by a called procedure.

<sup>2</sup> Annotation sets are treated as a separately defined class in the Architecture, but its `Nth` and `Length` operations are designed to parallel those of the other sets.

## 2.5 Error Handling

A number of operations in the architecture describe error conditions (generally with the phrase "it is an error if..."). Such errors should be implemented by signaling an error rather than by returning an error value (this could be performed in C by using the **longjmp** function and in Common Lisp by the **error** function). The C implementation provides utility routines which simplify the use of **longjmp** for this purpose.



## 3.0 BASIC CLASSES

### 3.1 Attributes

A number of classes will have "attributes". This is a list of feature-value pairs, where the feature names are arbitrary strings and the values can be any of a number of types:

#### Class Attribute

##### Properties

Name: string

Value: AttributeValue

##### Operations

CreateAttribute (name: string, value: AttributeValue): Attribute

#### Class AttributeValue

##### Properties

Value: string OR ObjectReference OR sequence of AttributeValue

##### Operations

CreateAttributeValue(string OR ObjectReference OR sequence of AttributeValue): AttributeValue<sup>3</sup>

TypeOf (AttributeValue): one of {**string, sequence, CollectionReference, DocumentReference, AnnotationReference, AttributeReference**}

returns a member of the enumerated type, indicating the type of AttributeValue

Note: AttributeValue is made a separate class, with an explicit TypeOf operator, out of deference to languages such as C without dynamic type identification. Because AttributeValue can take on multiple types, including types such as strings which would not use a generic container structure, implementations in such languages must provide an explicit type discriminator here, accessible through the TypeOf operator. The value of an attribute may be (*inter alia*) a reference to a collection, document, annotation, or attribute:

#### Abstract Class ObjectReference

#### Class CollectionReference

Type of ObjectReference

##### Properties

CollectionName: string

##### Operations

CreateCollectionReference (Collection): CollectionReference

#### Class Document Reference

Type of ObjectReference

##### Properties

CollectionName: string

DocumentId: string

##### Operations

CreateDocumentReference (Document): DocumentReference

CreateDocumentReferenceFromSpecification(ReferencedCollection:string, ReferencedDocumentId: string):DocumentReference

creates a new DocumentReference with CollectionName equal to ReferencedCollection and DocumentId equal to ReferencedDocumentId.

---

<sup>3</sup> For implementation in languages which cannot determine the type of the argument at run time, such as C, this operation requires two arguments. The additional argument (the first of the two arguments) is of enumerated type "one of {string, sequence, CollectionReference, DocumentReference, AnnotationReference, AttributeReference}" and specifies the type of the second argument, which is the value itself.

### Class **AttributeReference**

Type of ObjectReference

#### Properties

CollectionName: string

DocumentId: string

AttributeName: string

#### Operations

CreateAttributeReference (Document, AttributeName: string): AttributeReference

### Class **AnnotationReference**

Type of ObjectReference

#### Properties

CollectionName: string

DocumentId: string

AnnotationId: string

#### Operations

CreateAnnotationReference (Document, Annotation): AnnotationReference

ObjectReferences are references to (names of) persistent collections, documents, etc., and not to the object instances created by opening a collection, etc. It is therefore possible to have ObjectReferences to documents in collections which are not currently open; it is even possible to have references to documents which have been deleted from a collection. Because of the variety of objects which can be referenced, the Architecture does not provide a single dereferencing operator. Dereferencing must be done explicitly by the Application using the property accessors — opening the collection, accessing the document, accessing the annotation in the document, etc.

An abstract class for objects which have attributes is defined as:

### Abstract Class **AttributedObject**

#### Properties

Attributes: sequence of Attribute

#### Operations

PutAttribute (AttributedObject, name: string, value: AttributeValue)

assign *value* as the current value of attribute *name* of *object*, overwriting any prior assignment of a value to that attribute

GetAttribute (AttributedObject, name: string): AttributeValue OR *nil*

if attribute *name* of *object* has been assigned a value by a prior PutAttribute operation, return that value, else return *nil*

RemoveAttribute(AttributedObject, name: string)

if AttributedObject has an Attribute whose Name property is name, remove that attribute from AttributedObject (otherwise do nothing)

## 3.2 Persistent Objects

The TIPSTER Architecture assumes a name space of persistent objects; each persistent object is assigned a unique name (a string). If the Architecture is operating in a networked environment, this name will presumably consist of a host name and a unique name on that host.

The (abstract) class *Persistent Object* is introduced, which is a superclass of any class of persistent objects.

### Abstract Class **PersistentObject**

#### Properties

Name: string

#### Operations

Create.PersistentObject (name: string): PersistentObject

creates a new object of a specified class, and returns that object (it is an *error* if *name* is the name of an existing persistent object)

Open.PersistentObject (name: string): PersistentObject

*name* should be the name of an object of class PersistentObject, created by a prior Create.PersistentObject operation; the object with that name is returned

Close (object: PersistentObject)

    frees any local memory associated with this object *the Architecture assumes that all Persistent Objects will be automatically closed on system termination*

Sync (object: PersistentObject)

    saves any changes made to *object* in persistent storage

Destroy (name: string)

    erases the persistent instance of the object (it is an *error* if *name* is not the name of a persistent object)

The architecture does not require us to identify persistent object names with file names, but this may be the simplest way to manage initial implementations. In the present architecture DocumentCollectionIndexes and QueryCollectionIndexes are persistent; Collections are optionally persistent (Documents are not persistent objects themselves but have persistence as a part of a Collection).

### 3.3 Byte Sequences

The decision about the representation of a sequence of bytes, which constitutes the contents of a document, should be hidden from most applications. To do so, the class ByteSequence is introduced. The minimal requirement for an implementation of the Architecture is to be able to obtain the length of a ByteSequence, and to convert between a ByteSequence and a string:

#### Class ByteSequence

##### Operations

Length (ByteSequence): integer

    returns the number of bytes in ByteSequence

ConvertToString (ByteSequence): string

CreateByteSequence (string): ByteSequence

(In fact, the simplest implementation of a ByteSequence will probably be as a string, so the conversion will be an identity operation.) Implementations may choose to supplement these with additional operations for creating and accessing ByteSequences, for two reasons:

1. For applications involving large documents, the implementation may wish to provide the ability to directly access portions of the document. This may be done through operations which retrieve substrings of a ByteSequence, or through operations which allow a ByteSequence to be opened to a stream (for subsequent read and write operations).
2. A collection of documents needs to be converted into a TIPSTER Collection prior to processing within the Architecture. For large collections which are already in place on some data store, such as a file system or a data base, it may be highly desirable to create the TIPSTER Collection without copying the document text. A TIPSTER implementation can support this capability by allowing a ByteSequence to be created as a reference to a portion of this data store. For example, the implementation could define a "file segment" as a portion of a file (with start and end positions), and support operations for creating a ByteSequence from a file segment. Alternatively, an application based on a data base could define an operation for creating a ByteSequence from a data base field.



## 4.0 DOCUMENTS AND COLLECTIONS

### 4.1 Documents

The document is the central object class in the TIPSTER architecture. As a unit of information, it serves several basic functions within the architecture:

- it is the repository of information about a text, in the form of attributes and annotations (although annotations will in general refer to portions of documents)
- it is the atomic unit in building collections
- it is the atomic unit of retrieval in detection operations

Each Document is part of one or more Collections (see Section 4.2). A Document has persistence by virtue of being a member of a Collection, and can be accessed only as a member of a Collection. Each document is given a unique identity by its *Id* property, which is copied by the CopyDocument and CopyBareDocument operations, and is also copied when a new collection is created by document retrieval operations.

A Document may have a BaseDocument Attribute, which identifies a Document from which this Document was derived. Those Tipster operations that use a Document's RawData component should, if the RawData component is nil, use instead the RawData component of the Document's BaseDocument. Such operations include DocumentCollectionIndex.Augment, QueryCollectionIndex.RetrieveQueries, Document.Annotate, Document.WriteSgml, and Collection.AnnotateCollection. If the Document's RawData is nil and there is no BaseDocument, then these operations should ignore the Document.

Each document is part of one or more Collections (see Section 4.2). A Document has persistence by virtue of being member of a Collection, and can be accessed only as a member of a Collection. Each document is given a unique identity by its Id component and its native Collection: an Id is unique within a Collection

### Class Document

**Type of** AttributedObject

#### Properties

Parent: Collection

the Collection of which this Document is a member;

Id: string

an internal document identifier, assigned automatically when a new Document is created, which is unique within the Collection in which the Document resides

ExternalId: string (**R, W**)

a document identifier assigned by the application

RawData: ByteSequence OR nil

the contents of the document prior to any TIPSTER processing. The byte-sequence may include subsequences representing text in multiple languages, as well as non-text material such as pictures, audio, and tables

Annotations: AnnotationSet

information about portions of the document (information about the document as a whole is stored in Attributes; a Document inherits an Attributes property by virtue of being a type of Attributed Object). Annotations may contain information about a Document related to the current Document. The knowledge of the relationship between Documents must be maintained by the application (possibly using Attributes).

#### Operations

CreateDocument (Parent: Collection, ExternalId: string, RawData: ByteSequence OR nil, annotations: AnnotationSet, attributes: sequence of Attribute): Document

creates a new document within the Collection Parent and assigns the document a new unique Id

CopyBareDocument (NewParent: Collection, Document): Document

makes a copy of *Document*, including only its *ExternalId*, and *RawData*, assigns a new unique *Id* to the copy and places the copy in collection *NewParent*. The attributes and annotations of the original document are not copied by this operation.

**CopyDocument** (*NewParent*: Collection, *Document*): Document

makes a copy of *Document*, including its *ExternalId*, *RawData*, attributes, and annotations, assigns a new unique *ID* to the copy, and places the copy in collection *NewParent*.

**Annotate** (*Document*, *AnnotatorName*: string)

invokes annotation procedure *AnnotatorName* on the *Document*; see Section 5.6.

**WriteSGML** (*Document*, *AnnotationSet*, *AnnotationPrecedence*: sequence of string): string

Converts a document together with a set of *Annotations* into SGML format. *AnnotationPrecedence*, which is a list of annotation types, is used to resolve conflicts when two annotations cover the same span: the tag corresponding to the annotation type which appears first in the list is written out first. The resulting document is in a "normalized" SGML, with all attributes and end tags explicit.<sup>4</sup>

**ReadSGML** (string, *Parent*: Collection, *ExternalId*: string): Document

Reads a string marked up with "normalized" SGML, with all attributes and end tags explicit, and generates a *Document* with the specified *ExternalId*, no attributes, and an *AnnotationSet* containing one annotation for each SGML text element marked in the input text. If the input violates these constraints (e.g., unmatched start tags) or violates SGML syntax (e.g., unmatched quotation marks within tags), an error will be signaled.<sup>5</sup>

As noted earlier, new sources of data will need to be converted by the application into Collections of Documents before they can be processed within the TIPSTER Architecture. The functions which perform these conversions will necessarily be specific to the type of data source, and hence a TIPSTER application will be required to provide these conversion operations when a new type of data source is to be used.

## 4.2 Collections

Documents are gathered into Collections, which may have attributes on the collection level as well as on the individual documents. Collections provide a permanent repository for documents within the TIPSTER Architecture.

Collections in general are persistent and hence have names; however, the Architecture also provides for volatile, unnamed Collections.

### Class Collection

Type of PersistentObject, AttributedObject

#### Properties

Owner: string (**R**, **W**)

Contents: sequence of Document ( $\mathbb{R}$ )

#### Operations

**CreateCollection** (name: string, attributes: sequence of Attribute): Collection

creates a named, persistent collection

**CreateVolatileCollection** (attributes: sequence of Attribute): Collection

creates an unnamed, volatile collection

**Length** (Collection): integer

returns the number of documents in a collection

**RemoveDocument** (Collection, *Id*: string)

removes the Document with *Id Id* from the Collection; if no such Document is present, the Collection is unchanged

**GetDocument** (Collection, *Id*: string): Document OR *nil*

returns the Document in the Collection with the given *Id*, or *nil* if no such Document exists

---

<sup>4</sup> The specification of this operation is subject to revision based on the experience of implementors in using these SGML representations in applications.

<sup>5</sup> The specification of this operation is subject to revision based on the experience of implementors in using these SGML representations in applications.

FirstDocument (Collection): Document OR nil

returns the `first' document within a collection and initializes data structures internal to the collection so that NextDocument can be used to iterate through the documents in a collection. Returns **nil** if no documents are found in the collection.

NextDocument (Collection): Document OR nil

returns the `next' document within a collection. Normally used to iterate through all documents in a collection. Returns **nil** if no more documents are found in the collection. FirstDocument and NextDocument must be well behaved in the presence of calls to CreateDocument and RemoveDocument. This means that a loop using FirstDocument and NextDocument must visit all documents which were in the collection when FirstDocument was called if and only if the documents are not deleted before the loop reaches them. Documents added after FirstDocument is called may or may not be encountered during the loop.

GetByExternalId (Collection, ExternalId: string): Document OR *nil*

returns the Document in the Collection with the given *ExternalId*; if several Documents have the same ExternalId, returns one of them; if none have this ExternalId, returns *nil*

AnnotateCollection (which: Collection, destination: Collection, AnnotatorName: string)

invokes annotation procedure *AnnotatorName* on a subset of Collection *destination*; see Section 5.6 for further information



## 5.0 DOCUMENT ANNOTATIONS: GENERAL STRUCTURE

Annotations, along with attributes, provide the primary means by which information about a document is recorded and transmitted from module to module within a system. This chapter elaborates the general structure of annotations, noting some of the issues which arise at each stage.

### 5.1 What Is Annotated?

An annotation provides information about a portion of the document (including, possibly, the entire document). The portion of the document is specified by a set of spans. Each span consists in turn of a pair of integers specifying the starting and ending byte positions in the RawData of the document (with the first byte of the document counting as byte 0).

#### Class Span

##### Properties

Start: integer

End: integer

##### Operations

CreateSpan (start: integer, end: integer): Span

*The current span design is intended for character-based text documents which may contain additional types of information such as graphical images, audio, or video, which needs to be retained and displayed, but which would not be further processed by components of the TIPSTER Architecture. For documents which do not contain text in the form of a sequence of characters, the meaning of a span will not necessarily be compatible with this start byte/end byte convention. For instance, in compressed video, the information contained in a sequence of frames cannot be located using starting and ending byte. Similarly, in a graphical image of a document, such as a fax, the most natural definition of a primitive subimage is likely to be a rectangle. Note that the data in a fax is not even byte aligned. All of these considerations indicate that eventually an opaque type for spans with a subclass being TextSpan will be needed.*

Most annotations will be associated with a single contiguous portion of the text, and hence with a single span. However, a set of spans is provided for in order to be able to refer to non-contiguous portions of the text. For example, an event might be described at the beginning of an article and again later in the article, but not in the intervening text; using a set of spans allows us to have an annotation for the event refer to these two passages. It would also allow for discontinuous linguistic elements, such as verb plus particle pairs ("I gave my gun up.").

#### 5.1.1 Code Sets and Character Positions

Positions in the RawData are represented internally in terms of byte offsets, rather than characters. This is necessary because the RawData may contain non-text data, such as graphics or sounds, for which character addressing would not be meaningful. However, once a text has been segmented into text and non-text portions, and the text portion into segments involving different character codes, it should be possible to provide operations at the character level (i.e., operations which are sensitive to the different sizes of characters in different codes). This segmentation into regions using different character code sets is to be recorded in the TIPSTER Architecture as Annotations on the document (see Section 6.1). By accessing these Annotations, an application can determine the code set employed at a specific position in a document, and hence the size of the character at that position. This information can be used to implement operations to extract a single character or advance to the next character position.

*More work is required on the multi-lingual design of the Architecture before such operations can be incorporated into the Architecture itself.*

#### 5.1.2 Modification of the Text

To allow annotations to modify the text (and, in particular, to insert characters) in such a way that subsequent accesses to the text see the modified text in place of the original text, it is necessary to require a representation of positions in a document which allows for insertions (e.g., by using integers above the length of the original string to refer to inserted elements). The current architecture does not allow for such changes; corrections to the text must be recorded as attributes on text elements which are explicitly accessed by subsequent processes. Alternatively, the application can create a new Document with a new RawText property which incorporates these modifications.

## 5.2 Information Associated With an Annotation

An annotation associates a *type* with a span of the document. Examples of possible types are *token*, *sentence*, *paragraph*, and *dateline*. In addition, one or more attributes may be assigned to each annotation.

### Class Annotation

**Type of** AttributedObject

#### Properties

Id: string

the identifier of an Annotation, which is nil when the Annotation is created and which is set when the Annotation is added to a Document; the value assigned is unique among the Annotations on that Document.

Type: string

Spans: sequence of Span

#### Operations

CreateAnnotation (Type: string, Spans: sequence of Span, attributes: sequence of Attribute): Annotation

Examples of simple attributes on annotations (attributes whose values are single strings) include a *type-of-name* attribute on name annotations, which might take on such values as "person", "country", "company", etc.; a *pos* (part of speech) attribute on *token* annotations, which might take on the Penn Tree Bank values, such as "NNS" and "VBG", and a *root* attribute on token annotations, which would record the root (uninflected) form of a token.

An example of an attribute whose value is another annotation would be a coreference pointer. An even more complex attribute value would be a template object, which may in turn contain pointers to several other annotations (for the text elements filling various slots in the template object).

## 5.3 Accessing Annotations

Because annotations are central to the TIPSTER architecture, it is expected that applications will have frequent need to access, search, and select annotations on a document. To meet this need, the Architecture defines a class AnnotationSet and a number of operations operating on such sets of annotations. In particular, operations are provided to support the sequential scanning of a document (AnnotationsAt, NextAnnotations) and to support the extraction of annotations meeting certain criteria (SelectAnnotations).

Although AnnotationSets are logically just sets of annotations, and could be implemented like other sets (e.g., as lists), a special class is provided in the expectation that implementations may wish to choose a more elaborate implementation (such as a sorted list or tree with one or more indexes) in order to implement the operations more efficiently.

Each Document includes as one property an AnnotationSet, holding the annotations on that Document. Most of the operations on AnnotationSets can also be applied to Documents, and in that case apply the same operation to the AnnotationSet property of the Document.

### Class AnnotationSet

#### Properties

Members: sequence of Annotation ( $\mathbb{R}$ )  
the individual annotations in the set

#### Operations

CreateAnnotationSet (): AnnotationSet  
creates an empty AnnotationSet

AddAnnotation (Document, Annotation): string  
adds an annotation to a document. If the *Id* slot of *Annotation* is *nil*, this operation creates a new annotation *Id* (unique for this document) and assigns it to the *id* field of *Annotation*. If the *Id* field of *Annotation* is filled (not *nil*), and there is an existing annotation on the document with the same *Id*, the new annotation replaces the existing annotation. The *Id* field of the annotation is returned.

RemoveAnnotation (Document OR AnnotationSet, Id: string)  
removes the annotation with the specified *Id* from the Document or AnnotationSet. It is an error if the document does not have an annotation with that *Id*.

- GetAnnotation (Document OR AnnotationSet, Id: string): Annotation  
 returns the annotation whose id slot is equal to the desired value. It is an error if no annotation has the specified identifier.
- Length (AnnotationSet): integer  
 returns a count of the number of annotations in *AnnotationSet*
- Nth (AnnotationSet, n: integer): Annotation  
 returns the *Nth* annotation in *AnnotationSet*, where the first annotation has index 0
- SelectAnnotations (Document OR *AnnotationSet*, type: string OR *nil*, constraint: sequence of Attribute): AnnotationSet  
 returns the (possibly empty) set of annotations from the Document or *AnnotationSet* which are of type *type* and which satisfy *constraint*. *constraint* is a sequence of attributes, where the *i*th attribute has name *a<sub>i</sub>* and value *v<sub>i</sub>*. An annotation satisfies the constraint if (for each *i*), attribute *a<sub>i</sub>* of the annotation has value *v<sub>i</sub>*. If *constraint* is the empty sequence, no constraint is placed on the attributes: all annotations of the given type are selected. If *type* is *nil*, annotations of all types satisfying the attribute constraints are included.
- DeleteAnnotations (Document OR *AnnotationSet*, type: string OR *nil*, constraint: sequence of Attribute)  
 removes from the Document or *AnnotationSet* all annotations which are of type *type* and which satisfy *constraint*. These arguments have the same significance as for *SelectAnnotations*, above.
- AnnotationsAt (Document OR *AnnotationSet*, Position: integer): AnnotationSet  
 returns the set of annotations from *Document* or *AnnotationSet* which start at the specified position
- NextAnnotations (Document OR *AnnotationSet*, Position: integer): AnnotationSet  
 Returns the set of annotations from *Document* or *AnnotationSet* which have the smallest starting point that is greater than or equal to *Position*.
- MergeAnnotations (*AnnotationSet*, *AnnotationSet*): *AnnotationSet*  
 returns the union of the Annotations in the two *AnnotationSets*

## 5.4 Annotation Type Declarations

A central goal in creating the TIPSTER architecture is for different modules to be able to share information about a document through the use of annotations. Such information sharing will be workable only if there are precise, formal descriptions of the structure of these annotations, and if the modules which create annotations adhere to these descriptions.

Therefore, *annotation type declarations* are introduced here which serve to document the information associated with different types of annotations. In the present architecture these declarations only serve as documentation; future generations of the architecture may seek to do type checking based on these declarations (see Appendix A.1).

Type declarations are organized into *packages*. A package will typically include a set of related annotation types. For example, a package may declare all the types of annotations used to represent the document structure for one message format (header, dateline, author, etc.). Another package, associated with an extraction system, would represent the annotation types corresponding to the template objects created by that system.

The declaration of a package of annotation types would consist of a package name declaration followed by one or more annotation type declarations. The package name declaration has the form

**type package** *identifier*

An annotation type declaration defines an annotation type; it specifies the attributes which such annotations may have and the type of value of each attribute. The declaration has the form

**annotation type** *identifier* ( *attribute-spec<sub>1</sub>* *attribute-spec<sub>2</sub>*, .. . );

where each attribute specification, *attribute-spec<sub>i</sub>*, has the form

*attribute-name*: *type-spec*

The *type-spec* specifies the type of allowable values of the attribute. The type spec may specify a basic type:

**Boolean**  
**string**  
**integer**  
**real**  
**document**  
**collection**

**annotation** (a reference to an annotation of any type)

it may specify an enumerated type by giving its alternative values:

( *value<sub>1</sub>*, *value<sub>2</sub>* ...)

it may specify a union of types by listing the alternative types:

( *type<sub>1</sub>* or *type<sub>2</sub>* or ...)

to indicate that the value may be of any one of the types listed; it may specify a compound type, either

**sequence of type**

which allows for a sequence of zero or more instances of type *type*, or

**optional type**

whose value may be either of *type* or be *nil*. Finally, *type-spec* may be a previously defined annotation type, specifying a reference to an annotation of that type.

One or more white-space characters (blanks, tabs, or newlines) are required between successive identifiers and alphabetic names; zero or more white-space characters are allowed before and after the separator characters " : ; ()". Any text between a left bracket and a right bracket ([...]) is considered a comment.

Here is a simple example based on the mini-MUC organization template (more elaborate template examples are given in Section 8):

type package organizations;

```

annotation type organization {org_name:    string,
                               org_aliases:  sequence of string,
                               org_type:     {government, company, other},
                               org_location:  sequence of typed_location};
annotation type typed_location {location:   string;
                                type:        {country city landregion province
                                              waterregion address oth-unk}};
```

## 5.5 Examples of Annotations

This section shows some simple examples of annotated documents. Each example is shown in the form of a table. At the top of the table is the document being annotated; immediately below the line with the document is a ruler showing the position (byte offset) of each character. Underneath this appear the annotations, one annotation per line. For each annotation is shown its Id, Type, Span, and Attributes. Integers are used as the annotation Ids. Also, for simplicity only a single Span for each Annotation is shown. The attributes are shown in the form *name = value*.

At the end of this section the type declaration packages which would be used to describe these annotations is shown.

The first example shows a single sentence and the result of three annotation procedures: tokenization with part-of-speech assignment, name recognition, and sentence boundary recognition. Each token has a single attribute, its part of speech (pos), using the tag set from the University of Pennsylvania Tree Bank; each name also has a single attribute, indicating the type of name: person, company, etc.

| Text                    |          |            |          |                  |
|-------------------------|----------|------------|----------|------------------|
| Cyndi savored the soup. |          |            |          |                  |
| 0... 5... 10.. 15.. 20  |          |            |          |                  |
| Annotations             |          |            |          |                  |
| Id                      | Type     | Span Start | Span End | Attributes       |
| 1                       | token    | 0          | 5        | pos=NP           |
| 2                       | token    | 6          | 13       | pos=VBD          |
| 3                       | token    | 14         | 17       | pos=DT           |
| 4                       | token    | 18         | 22       | pos=NN           |
| 5                       | token    | 22         | 23       |                  |
| 6                       | name     | 0          | 5        | name_type=person |
| 7                       | sentence | 0          | 23       |                  |

Annotations will typically be organized to describe a hierarchical decomposition of a text. A simple illustration would be the decomposition of a sentence into tokens. A more complex case would be a full syntactic analysis, in which a sentence is decomposed into a noun phrase and a verb phrase, a verb phrase into a verb and its complement, etc. down to the level of individual tokens. Such decompositions can be represented by annotations on nested sets of spans. Both of these are illustrated in our second example, which is an elaboration of our first example to include parse information. Each non-terminal node in the parse tree is represented by an annotation of type *parse*.

| Text                    |          |            |          |                                   |
|-------------------------|----------|------------|----------|-----------------------------------|
| Cyndi savored the soup. |          |            |          |                                   |
| 0... 5... 10.. 15.. 20  |          |            |          |                                   |
| Annotations             |          |            |          |                                   |
| Id                      | Type     | Span Start | Span End | Attributes                        |
| 1                       | token    | 0          | 5        | pos=NP                            |
| 2                       | token    | 6          | 13       | pos=VBD                           |
| 3                       | token    | 14         | 17       | pos=DT                            |
| 4                       | token    | 18         | 22       | pos=NN                            |
| 5                       | token    | 22         | 23       |                                   |
| 6                       | name     | 0          | 5        | name_type=person                  |
| 7                       | sentence | 0          | 23       | constituents= [1],[2],[3].[4],[5] |
| 8                       | parse    | 0          | 5        | symbol="NP",constituents= [1]     |
| 9                       | parse    | 14         | 22       | symbol="NP",constituents=[3],[4]  |
| 10                      | parse    | 6          | 22       | symbol="VP",constituents=[2],[9]  |
| 11                      | parse    | 0          | 22       | symbol="S",constituents=[8],[10]  |

In most cases, the hierarchical structure could be recovered from the spans. However, it may be desirable to record this structure directly through a *constituents* attribute whose value is a sequence of annotations representing the immediate constituents of the initial annotation. For the annotations of type *parse*, the constituents are either non-terminals (other annotations in the parse group) or tokens. For the sentence annotation, the constituents attribute points to the constituent tokens. A reference to another annotation is represented in the table as "[*Annotation Id*]"; for example, "[3]" represents a reference to annotation 3. Where the value of an attribute is a sequence of items, these items are separated by commas. No special operations are provided in the current architecture for manipulating constituents.

At a less esoteric level, annotations can be used to record the overall structure of documents, including in particular documents which have structured headers, as is shown in our third example<sup>6</sup>:

<sup>6</sup> Incounting characters, count one character for the *newline* between lines

| <i>Text</i>                                                                                                                                                                                                                                                                                                                                                                                                    |           |            |          |               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|------------|----------|---------------|
| To: All Barnyard Animals<br> 0... 5... 10... 15... 20...<br>From: Chicken Little<br> 25... 30... 35... 40... 45...<br>Date: November 10, 1194<br>... 50... 55... 60... 65...<br>Subject: Descending Firmament<br> 70... 75... 80... 85... 90... 95...<br>Priority : Urgent<br>. 100... 105... 110... 115...<br>The sky is falling. The sky is falling.<br>... 120... 125... 130... 135... 140... 145... 150... |           |            |          |               |
| <i>Annotations</i>                                                                                                                                                                                                                                                                                                                                                                                             |           |            |          |               |
| Id                                                                                                                                                                                                                                                                                                                                                                                                             | Type      | Span Start | Span End | Attributes    |
| 1                                                                                                                                                                                                                                                                                                                                                                                                              | Addressee | 4          | 24       | ddmmyy=101194 |
| 2                                                                                                                                                                                                                                                                                                                                                                                                              | Source    | 31         | 45       |               |
| 3                                                                                                                                                                                                                                                                                                                                                                                                              | Date      | 53         | 69       |               |
| 4                                                                                                                                                                                                                                                                                                                                                                                                              | Subject   | 78         | 98       |               |
| 5                                                                                                                                                                                                                                                                                                                                                                                                              | Priority  | 109        | 115      |               |
| 6                                                                                                                                                                                                                                                                                                                                                                                                              | Body      | 116        | 155      |               |
| 7                                                                                                                                                                                                                                                                                                                                                                                                              | Sentence  | 116        | 135      |               |
| 8                                                                                                                                                                                                                                                                                                                                                                                                              | Sentence  | 136        | 155      |               |

If the *Addressee*, *Source*, ... annotations are recorded when the document is indexed for retrieval, it will be possible to perform retrieval selectively on information in particular fields.

Our final example involves an annotation which effectively modifies the document. The current architecture does not make any specific provision for the modification of the original text. However, some allowance must be made for processes such as spelling correction. This information will be recorded as a *correction* attribute on *token* annotations and possibly on name annotations:

| <i>Text</i>                                                          |       |            |          |                              |
|----------------------------------------------------------------------|-------|------------|----------|------------------------------|
| Topster tackles 2 terrorbytes.<br> 0... 5... 10... 15... 20... 25... |       |            |          |                              |
| <i>Annotations</i>                                                   |       |            |          |                              |
| Id                                                                   | Type  | Span Start | Span End | Attributes                   |
| 1                                                                    | token | 0          | 7        | pos=NP correction=TIPSTER    |
| 2                                                                    | token | 8          | 15       | pos=VBZ                      |
| 3                                                                    | token | 16         | 17       | pos=CD                       |
| 4                                                                    | token | 18         | 29       | pos=NNS correction=terabytes |
| 5                                                                    | token | 29         | 30       |                              |

The sample annotations shown here would require the following type declarations:

type package basic;

annotation type token: {pos: string, correction: optional string};

annotation type name: {name\_type: {person, organization, other}};

annotation type sentence: {constituents: optional sequence of token};

type package parse;

annotation type parse: {symbol: string,  
constituents: sequence of (parse or token or name)};

type package message;

annotation type addressee;

annotation type source;

annotation type date: {ddmmyy: string};

annotation type subject;

annotation type priority;

annotation type body;

## 5.6 Invoking Annotators

Each TIPSTER system will be provided with a number of "annotators" procedures for generating annotations. There will be annotators for different types of annotations; for example, for tokenization, for sentence segmentation, for name recognition, etc. In addition, there may be multiple annotators of a single type; e.g., multiple tokenizers.

Each annotator is assigned a name (a string). It is invoked by

Annotate (Document, AnnotatorName: string)

or

AnnotateCollection (which: Collection, destination: Collection, AnnotatorName: string)

The first form annotates a single Document. The second form annotates a Collection or a subset thereof. This uses Collection *which* to determine which documents to process, and Collection *destination* to record the annotations. For each document in collection *which*, if the same document (a document with the same Id) appears in *destination*, annotate that document in collection *destination*. This calling sequence allows us to selectively apply annotators to subsets of a collection, but to keep all the annotations together in the "original" collection. If *which* and *destination* are identical, the entire collection is annotated.

*Note: Future versions of the architecture will include operations for managing the set of annotators: for adding an annotator to the set of annotators, for recording the types of annotations produced by an annotator, and for searching the set of annotators.*

## 5.7 External Representation of Annotations

The TIPSTER architecture provides an external, character-based representation of annotated documents, so that such documents can be interchanged among modules (possibly as part of different TIPSTER systems on different machines) without regard to the internal representation used on particular machines. A representation based on SGML has been selected in order to be able to make use of the large number of existing applications which can operate on SGML documents.

In this representation, if the document consists of the text "aaaa bbbb cccc", and the span corresponding to "bbbb" has been assigned an annotation of type *atype* with id *ident*, and this annotation has attributes *attribute1*, *attribute2*, ... with values *value1*, *value2*, ... then the external representation of the annotated document will be

```
aaaa <atype id=ident attribute1=value1 attribute2=value2 . . . >bbbb</atype> cccc
```

This representation is produced by the *WriteSGML* operation, which takes as arguments a document, an AnnotationSet, and a precedence list among annotation types. This precedence list is used to determine the nesting of SGML tags if two annotations involve the same span. A complementary operation, *ReadSGML*, reads a SGML document which conforms to this format (with all attributes and end tags explicit) and creates a document with annotations.

*The specification of these operations is subject to revision based on the experience of implementors in using these SGML representations in applications.*

*It may be desirable to have a second external representation which much more closely parallels the internal property structure of the annotations, particularly if annotations are to be exchanged over a network.*

## 5.8 Annotation Schemata and Style Sheets

Different groups of annotations normally exist in some fixed structural relationships to one another. Thus, a text body may consist of paragraphs, a paragraph of sentences, a sentence of tokens, etc. For an SGML document, these relationships are provided by a DTD. At present, the Architecture includes a very limited amount of such information in the form of the PrecedenceList argument to *WriteSGML*; it may be desirable to include in later versions of the architecture an AnnotationSchema more analogous to a DTD.

When an SGML form is generated from an annotated document, rules must be applied to realize each type of annotation as a sequence of characters. In the present version, these rules are assumed to be built in to the *WriteSGML* operation, but in later versions it may be desirable to provide these rules explicitly as a StyleSheet. A TIPSTER System would have a default StyleSheet, but it may be necessary to extend the *WriteSGML* operation to use a different, explicitly specified style sheet.



## 6.0 TYPES OF DOCUMENT ANNOTATIONS

### References:

- (1) **Guidelines for Electronic Text Encoding & Interchange (TEI)**,  
<http://etext.virginia.edu/TEI.html>
- (2) **ISO 639**, <http://etext.lib.virginia.edu/tei/iso639.html>
- (3) **ISO 3166**, <http://crl.nmsu.edu/Research/Projects/tipster/annotation/iso3166.html>
- (4) **ISO 8601**, <http://crl.nmsu.edu/Research/Projects/tipster/annotation/iso8601.html>
- (5) **Corpus Encoding Standard (CES)**, <http://www.cs.vassar.edu/CES/>
- (6) **Z39.50**, <http://lcweb.loc.gov/z3950/agency/agency.html>

The TIPSTER Architecture defines some standard annotations and associated attributes. If these names are used they must be used as defined. Other annotations and attributes may be defined for a TIPSTER application. If well defined annotations are created and it is expected that they will have extended usage, they should be submitted, via an RFC, for inclusion in the Architecture.

The annotations and attributes described herein are based upon the Corpus Encoding Standard (CES). The notation used is different to avoid confusion with CES which follows the method of embedding the tags in the text whereas TIPSTER carries the annotation separate from the text and associates it to the text by the span.

### 6.1 Annotations for Major Document Elements

The follow sections describe annotations at the minimum encoding level required for Level 1 CES conformance, requiring markup for gross document structure (major text divisions), down to the level of the paragraph. The annotation types described below are: Doc, Header, Text, Division and P (Paragraph).

#### **annDoc**

CES requires markup to encode a single document. However the annotation will be redundant with the same data represented by a TIPSTER document object. The span of the annotation will be the entire span of the document. In addition to the global attributes for annotations only one attribute is listed:

#### **attDocref**

An attribute value is a Document Reference which contains a pointer to the TIPSTER document object. The common document attributes for this document could be obtained through this reference.

The corresponding CES tag is <cesDoc>

#### **annHeader**

The header is information about the document typically appearing at the beginning of the document. This information should be contained in the span of the Header annotation. This includes elements such as the document source, date, author, and title. Many of these sub-elements will also be encoded as annotations. See Section 6.5.

The corresponding CES tag is <cesHeader>

#### **annText**

The text annotation contains the span of text that make up the contents of a document. It includes everything except the header information. No other attributes in addition to the global attributes are defined for this annotation type.

The corresponding CES tag is <text>

#### **annDivision**

This is an optional element for the CES and marks any subdivision of a written text, e.g. chapter, section, sub-section, article. For TIPSTER the Division annotation has one additional attribute beyond the global attributes:

#### **attType**

The attribute value will be a String categorizing the division in some respect. The categories are initially: PART, SECTION and CHAPTER.

The corresponding CES tag is <div>

**annP**

A paragraph in a written text.

The corresponding CES tag is <p>

**6.2 Annotations for Document Header Elements**

These elements are standard annotations for text that normally occurs in the header section of the document. This is meta-information about the document maybe, but not always, contained in the text of the original document. If the information is NOT contained in the document the span values are set to -1. This is a simplified version of information required by the CES header. The CES header defines a more highly structured set of elements. The annotations described here are intended to fill those elements in the CES structure without modifying the structure of the original electronic text.

**annDocno**

A document identifier string in the text. This could be an ISBN number or some other string used to identify the document.

**annTitle**

The title of the work, including sub titles. For documents containing the title directly in the text.

**annAuthor**

Text referring to the author or authors of the document.

**annPubDate**

A calendar date for the document.

**attISO8601**

Contains the ISO 8601 normalized form of the date.

**annPublisher**

A proper name of a person, place or institution.

**annPubPlace**

The place of publication for the document.

**6.3 Structural Annotations**

The following annotations are suggested for marking document structure. These are primarily based on a set defined for the CES standard. Please refer to the CES documentation for further explanation.

Annotations for paragraph level elements:

**annP**

A paragraph in a written text; (mentioned in Section 6.1, shown here for completeness)

**annSp**

Contains spoken text.

**annCaption**

A heading or title attached to a picture or diagram.

**annQuote**

A quotation for some author other than that of the surrounding text.

**annList**

A collection of distinct items.

**annFigure**

The location of graphic data.

**annBibl**

A bibliographic citation.

**annNote**

Notes that are part of the original data (e.g., footnotes).

**annTable**

Contains text displayed in tabular form.

## 6.4 Sub-paragraph Annotations

The following annotations are suggested for marking document structure below the paragraph level. These are primarily based on a set defined for the CES standard. Please refer to the CES documentation for further explanation. The Named Entity elements defined for MUC-6 are shown with their equivalent CES elements.

### **annAbbr**

Contains an abbreviation. One attribute can be specified.

### **attExpan**

Contains the expansion of the abbreviation.

### **annDate**

Contains date data. The MUC equivalent for TIMEX of type DATE.

### **attISO8601**

Contains the ISO 8601 normalized form of the date.

### **annMeasure**

Text containing a quantity of some type. This is the MUC equivalent for NUMEX.

### **attType**

The type attribute should take one of the following values:

WEIGHT  
LENGTH  
COUNT  
AREA  
VOLUME  
CURRENCY  
TEMPERATURE  
PERCENT

### **annName**

A proper name. The MUC equivalent for ENAMEX.

### **attType**

The type of proper noun. The values are:

PERSON, (MUC equivalent PERSON\_NAME)  
PLACE, (MUC equivalent LOCATION)  
ORG, (MUC equivalent ORGANIZATION\_NAME)

### **annNumber**

Contains a number.

### **attNormal**

The normalized form of the number; the normalization function is application dependent.

### **annTerm**

A technical word or phrase.

### **annTime**

The time of day in any form. The MUC equivalent for TIMEX of type TIME.

### **attISO8601**

Contains the ISO 8601 normalized form of the date.

### **annForeign**

A section of text that is in a different language than the surrounding text. The global attLang attribute is used to specify the language.

## 6.5 Common Document Attributes

Five common attributes are defined for TIPSTER Documents. Note: The TIPSTER architecture defines an External ID that can be used for document identification. In many cases, the attributes defined below will

have values taken from the text of the original document. An annotation reference containing the value shall be used as the value of the attribute when it is available.

**attTitle**

See Section 6.2 for the definition. Repeated here for completeness.

**attDocno**

See Section 6.2 for the definition. Repeated here for completeness. In some TREC and MUC applications this can be equivalent to the External ID document property.

**attLang**

The attLang attribute is used to identify a primary language value for the document. The value should be either a two-letter code from ISO639 or the three letter code from ISO639-2. Further specification for country is possible with an extension taken from ISO3166, (e.g., en.us for English in the United States, en.gb for English in the United Kingdom). A similar alternative that can include language and script variants has been defined for HTML 4.0. See RFC1766 for ftp access.

**attCharset**

A string specifying the character set used to encode the contents of the document, e.g., ISO-8859-1.

**6.6 Common Annotation Attributes**

Five global attributes are defined for TIPSTER Annotations. Two of these attributes, attLang, and attCodeset, are also used as global document attributes. The document version sets an attribute value for the whole document. The annotation attribute can be used to override these values. Note that all annotations also have an ID property that is unique within a document.

**attAnnotator[required]**

A string that identifies the name of the process or user that created the annotation.

**attComment**

A string that can be used to record any general information about the annotation.

**attN**

A number or label that can be used by applications to order a sequence of annotations, e.g., paragraph number.

**attLang**

See Section 6.5 for the definition. Repeated here for completeness.

**attCharset**

See Section 6.5 for the definition. Repeated here for completeness.

**attAlt**

The attAlt attribute is a sequence of alternate annotations of either the same type that have different sub-spans (defining the MUC-6 ALT attribute), or annotations that have the same span but are of different annotation types or attributes (defining alternate TYPE attribute values). These are used for generating answer key templates or markup.

**attStatus**

The attStatus attribute directly maps to the MUC-6 STATUS attribute used for optional markup. The value of this attribute can be either null or the STRING "OPT".

**6.7 Example of Annotations and Attributes**

0            10            20            30            40  
 This table presents an example of applying Ann. to a piece of  
           50            60            70            80            90            100  
 text. The example is from the number 6.7 to the end of the  
           110            120            130            140            150            160  
 table caption.  
           170            177

| ID | Type     | Span Start | Span End | Attributes |
|----|----------|------------|----------|------------|
| 1  | annTitle | 0          | 41       |            |
| 2  | annText  | 0          | 310      |            |
| 3  | annP     | 42         | 177      |            |

|   |            |     |     |                     |
|---|------------|-----|-----|---------------------|
| 4 | annTable   | 178 | 299 |                     |
| 5 | annCaption | 300 | 310 |                     |
| 6 | annNum     | 144 | 146 |                     |
| 7 | annAbbr    | 86  | 89  | attExpan=Annotation |

Annotations  
300            310



## 7.0 DETECTION

### 7.1 Object Classes

#### 7.1.1 Detection Needs and Queries

The user's request for documents is initially prepared in the form of a `DetectionNeed`: a document with a variety of SGML-delimited fields. (See Appendix E for an alternate specification, DN2, which defined the DTD for a Detection Need) A `DetectionNeed` is a type of `Document`, and so partakes of all the operations which can be applied to `Documents`. As a specialization of `Collections`, the Architecture includes `DetectionNeedCollections`; these are required primarily for routing operations, which typically involve sets of `DetectionNeeds`.

The `DetectionNeed` is transformed in two stages: it is first transformed into a `DetectionQuery`, and thence into either a `RetrievalQuery` or a `RoutingQuery`. `DetectionNeeds` are independent of the specific retrieval engine employed, while `DetectionQueries`, `RetrievalQueries`, and `RoutingQueries` are specific to a particular retrieval engine. The `DetectionQuery` is specific to the retrieval engine but independent of the collection over which retrieval is to be performed, and the operation (retrieval or routing) to be performed; the `RetrievalQuery` and `RoutingQuery` are specific to the retrieval engine, to the operation, and to a collection (they may incorporate, for example, term weights based on the Inverse Document Frequencies in a collection). The transformation process is divided into these two stages because a retrieval system may provide specialized tools for modifying the `DetectionQuery`.

#### 7.1.2 Detection Needs

##### Class `DetectionNeed`

**Type of Document**

**Description**

A system-independent description of the contents of the documents that the user would like to retrieve. The description may be in natural language, expressed with query language operators (described below), or a combination of natural language and query language operators.

**Operations**

`QueryGenerator (DetectionNeed): DetectionQuery`

Generate a system-specific `DetectionQuery` from an analysis of the `DetectionNeed`; the `DetectionQuery` has the same `ExternalId` as the `DetectionNeed`.

Query language operators are represented within the `DetectionNeed` using SGML-style tags. Each query language operator has the following syntax.

**<OPERATOR> argument + </OPERATOR>**

That is, an operator consists of an operator field marker (e.g. **<OPERATOR>**), one or more arguments, and an ending field marker (e.g. **</OPERATOR>**). Operators may be nested arbitrarily. Operator characteristics can be altered as shown. When alternatives are given (e.g. **EXACT** or **FUZZY**), the first one listed is the default. The default value for numeric arguments is 1.

It is not necessary for a system to implement each operator exactly as described below. A compliant system is one that can translate any valid `DetectionNeed` into its own query language, and that documents how each operator is handled. A system may ignore operators that it does not implement, or it may map them to the nearest reasonable alternative in that system's query language.

Any text not explicitly encapsulated in a query language operator is assumed to be implicitly encapsulated by the **<SUM>** operator (described below).

When it is necessary to distinguish among two or more `DetectionNeeds`, for example when they are stored in an ASCII file, the **<DETECTION-NEED>** SGML tag indicates the beginning of a `DetectionNeed`, and the **</DETECTION-NEED>** SGML tag indicates the end of the `DetectionNeed`. Text that is not enclosed between these tags is handled in a system-dependent (i.e. not defined by the TIPSTER architecture) manner. The operators are listed below, in alphabetic order.

**<AND MATCH=[EXACT | FUZZY]>**

Document should contain all arguments. **EXACT** match means that each document must contain all of the arguments. **FUZZY** match means that a document may be returned if it lacks one or more arguments, but the document is presumably ranked lower than documents that match all arguments.

**<AND-NOT MATCH=[EXACT | FUZZY]>**

Document should contain the first argument but not the second. Only two arguments can be specified for this operator.

**<COMMENT>**

All tokens until **</COMMENT>** are comments, to be ignored when creating DetectionQuery objects.

**<DOC-ANNOTATION=name>**

The arguments are to be matched against that portion of the document annotated with the annotation of type "name". Note that annotations may denote document structure, so that this operator may be used to restrict the match to within a single phrase, sentence, paragraph, section, etc.

**<DOC-ATTRIBUTE=name>**

The arguments are to be matched against the value of attribute "name".

**<NL>**

The arguments are a natural language description of part of the information need. No other query operator can occur in the **<NL>** description of the information need. (Any operators encountered are to be treated as text.) **</NL>** ends the field, unless it is escaped (see below).

**<ESCAPE>**

All tokens until **</ESCAPE>** are query terms, not operators. If the next token is **</ESCAPE>** then it is a query term, and not the end of the **<ESCAPE>**.

**<NONRELEVANT>**

The arguments are the Ids of documents that the user has judged to be not relevant to the information need.

**<OR MATCH=[EXACT | FUZZY]>**

Document should contain at least one argument.

**<PARAGRAPH MATCH=[EXACT | FUZZY], DISTANCE=n, [UNORDERED | ORDERED]>**

Document should contain all arguments within n paragraphs.

**<PHRASE MATCH=[EXACT | FUZZY], DISTANCE=n, [UNORDERED | ORDERED]>**

Document should contain all arguments within n phrases.

**<RELEVANT>**

The arguments are the Ids of documents that the user has judged to be relevant to the information need.

**<SENTENCE MATCH=[EXACT | FUZZY], DISTANCE=n, [UNORDERED | ORDERED]>**

Document should contain all arguments within n sentences.

**<SUM>**

Functionally, this operator is like an **<OR>** operator: The document must contain one or more arguments. However, the user may assume that documents that match more arguments are generally ranked higher than documents that match fewer arguments. (Typically used with vector-space or probabilistic systems.)

**<SYNONYM>**

The arguments are considered synonyms.

**<WEIGHT n>**

Applies a weight of n to its argument. May affect a document's score, depending upon the retrieval algorithm used.

**<WORDS MATCH=[EXACT | FUZZY], DISTANCE=n, [UNORDERED | ORDERED]>**

Document should contain all arguments within n words.

An SGML-like syntax was chosen because it is expressive, relatively easy to read, and system neutral. It allows the operator characteristics to be tailored without introducing a large number of special-purpose operators.

## **Class DetectionNeedCollection**

**Type of Collection**

**Description**

A Collection of Documents, all of which are DetectionNeeds.

### 7.1.3 Queries

#### Class **DetectionQuery**

**Type of Document**

**Description**

the system-specific (but collection-independent) translation of a *DetectionNeed*.

**Properties**

*DetectionNeed*: *DocumentReference*

a reference to the *DetectionNeed* from which this query is derived

**Operations**

*FormRetrievalQuery* (*DetectionQuery*, sequence of *DocumentCollectionIndex*): *RetrievalQuery*  
translate the *DetectionQuery* into an *RetrievalQuery* by using the information (e.g., document frequencies of terms, similarities between terms) in the set of *DocumentCollectionIndexes*; the *RetrievalQuery* has the same *ExternalId* as the *DetectionQuery*

*FormRoutingQuery* (*DetectionQuery*, sequence of *DocumentCollectionIndex*): *RoutingQuery*  
translate the *DetectionQuery* into a *RoutingQuery* by using the information (e.g., document frequencies of terms, similarities between terms) in the set of *DocumentCollectionIndex*; the *RetrievalQuery* has the same *ExternalId* as the *DetectionQuery*

*EditQuery* (*DetectionQuery*)

optional: this system-specific operation allows the user to modify the query, providing information which cannot be provided through the (system-independent) *DetectionNeed*

#### Class **RetrievalQuery**

**Description**

the translation of a *DetectionQuery* which is based on a particular *DocumentCollectionIndex* and intended for use in retrospective retrieval

**Properties**

*DetectionNeed*: *DocumentReference*

a reference to the *DetectionNeed* from which this query is derived

**Operations**

*ScoreDocuments* (*Collection*, *RetrievalQuery*)

assign to each *Document* in *Collection* an attribute *relevance* whose value indicates the relevance of the document to the query

*UpdateUsingRelevanceFeedback* (*RetrievalQuery*, *relevant\_docs*: *Collection*, sequence of *DocumentCollectionIndex*): *RetrievalQuery*

this operation updates the *RetrievalQuery* using relevance feedback, and returns the updated (or new) *Query*. The relevance feedback is provided through the *relevant\_docs* argument. Each document in this collection should have an Attribute *relevant* with the value "true" or "false". Furthermore, if that value is "true", the entry may also have one or more Annotations of type *relevant-section* whose Spans indicate the relevant sections of the document.

*RetrievalQueryFromRelevanceJudgements* (*relevant\_docs*: *Collection*, sequence of *DocumentCollectionIndex*, *DetectionNeed*): *RetrievalQuery*

this operation is similar to *Update UsingRelevanceFeedback*, but creates a new *RetrievalQuery* from scratch based on the relevance judgments recorded in *relevant\_docs*. The *DetectionNeed* parameter is required since each query must point to the original *DetectionNeed*; this *DetectionNeed* may contain a narrative characterization of the query being created, but no information from the *DetectionNeed* is used in creating the query

#### Class **RoutingQuery**

**Description**

the translation of a *DetectionQuery* which is based on a particular *DocumentCollectionIndex* and intended for use in routing.

### Properties

DetectionNeed: DocumentReference  
a reference to the DetectionNeed from which this query is derived

### Operations

UpdateUsingRelevanceFeedback (RoutingQuery, relevant\_docs: Collection, sequence of DocumentCollectionIndex): RoutingQuery  
RoutingQueryFromRelevanceJudgments (relevant\_docs: Collection, sequence of DocumentCollectionIndex, DetectionNeed): RoutingQuery  
these operations are exact analogs of the operations with the same names which apply to RetrievalQueries.

## 7.1.4 Document and Query Indexes

The TIPSTER Architecture provides for two types of document detection operations: retrieval and routing. In essence, retrieval involves the comparison of a single query against a large number of documents, while routing involves the comparison of a single document against a large number of queries (or "user profiles"). As a preliminary step for retrieval, generally, the set of documents must be pre-processed. Typically, this involves the creation of a term index, but it may also involve the gathering of various statistics about the set of documents (such as term document frequencies, term co-occurrence frequencies, and even term similarities based on co-occurrence). The result of all this preprocessing is a *DocumentCollectionIndex*. Retrieval is then performed by sending a query (in the form of an RetrievalQuery) to the DocumentCollectionIndex; the DocumentCollectionIndex returns a list of relevant documents.

### Class DocumentCollectionIndex

**Type of Persistent Object**

#### Description

a form of a Collection which is capable of responding to DetectionQuery. For most systems, this involves the annotation of the documents in the collection with approach-specific annotations, and then the creation of an inverted index involving these annotations. For some systems, however, an "index" might just be a normalized copy of the original text in a form which can be scanned by high speed search software. Also, indicates which Document terms and passages matched the RetrievalQuery

#### Operations

Augment (DocumentCollectionIndex, Collection)  
adds all the documents in Collection to the DocumentCollectionIndex. If a Document in the Collection has an Attribute with the name "BaseDocument", the DocumentCollectionIndex uses the Attribute's value as the "BaseDocument" value when the Document is returned as a result of a search.

RetrieveDocuments (sequence of DocumentCollectionIndex, RetrievalQuery, NumberToRetrieve: integer, MinThreshold: float, MaxThreshold: float Monitor or nil): Collection or nil)

returns a collection of Documents (of maximal length *NumberToRetrieve*) corresponding to those Documents in DocumentCollectionIndex that are most closely related to the DetectionNeed from which the Retrieval Query is derived. Each Document in the returned Collection will have a nil RawData property and an Attribute with name "BaseDocument" and, as value, a Document Reference identifying the corresponding Document in the DocumentCollectionIndex. All returned Documents will have a relevance rating above the specified MinThreshold and below the specified MaxThreshold. Threshold values will be in the range of 0.0 to 1.0. A MinThreshold of 0.0 and a MaxThreshold of 1.0 indicates that all matching Documents up to a maximum NumberToRetrieve will be returned. If no Documents match the RetrievalQuery argument or the MinThreshold/MaxThreshold criteria, nil will be returned.

The DocumentCollectionIndex will provide progress updates as requested by the Monitor. A nil argument means that no progress monitoring is required. A retrieval operation canceled by the Monitor object's MonitorProgress operation returns a Collection of accumulated documents.

MatchQueryTerms(sequence of DocumentCollectionIndex, RetrievalQuery, Document)

Document is a Document which has been returned in the Collection created by a RetrieveDocuments() operation.

RetrievalQuery is the query evaluated during the RetrieveDocuments() operation.

sequence of DocumentCollectionIndex is the set of indices against which the RetrievalQuery was evaluated.

The MatchQueryTerms() operation will add Annotations to the Document to indicate terms and passages in the Document which match the input RetrievalQuery.

annotation type RelevantPassage

{rank : String }

annotation type RelevantTerm

{rank : String }

The query term Annotations indicate where the RawData component of a particular Document matches the RetrievalQuery. Because these Annotations are specific to an instance of a RetrievalQuery they are not Intended to be a permanent part of the Document. Therefore, users may wish to delete these query term Annotations when they are finished reviewing their RetrieveDocuments results.

In routing, a set of queries or user profiles (in the form of RoutingQueries) are pre-processed to create a QueryCollectionIndex. Routing is then performed by sending a Document to a QueryCollectionIndex; what is returned is a set of relevant profiles (in the form of a DetectionNeedcollection).

## Class QueryCollectionIndex

Type of PersistentObject

### Description

the translation of a collection of RoutingQueries into a format which is efficient for performing document routing. Also, indicates which Document terms and passages matched the RoutingQuery.

The RetrieveQueries() operation returns only the Collection of DetectionNeeds matching a particular Document. To provide highlighting information for that <Document, DetectionNeed> pair, the Search Engine requires the RoutingQuery representation of the DetectionNeed which caused the Document to be matched. The GetQuery() operation allows the RoutingQuery matching the DetectionNeed to be retrieved from the QueryCollectionIndex and used later in the MatchQueryTerms() operation.

### Operations

AddQuery (QueryCollectionIndex, RoutingQuery, MinThreshold: float, MaxThreshold: float)

Adds RoutingQuery to the QueryCollectionIndex. If a RoutingQuery in the QueryCollectionIndex has a DetectionNeed component matching the DetectionNeed component of the RoutingQuery argument, the existing RoutingQuery is replaced by the RoutingQuery argument.

The MinThreshold and MaxThreshold arguments are used to control the Documents the RoutingQuery matches during RetrieveQueries() operations. The RoutingQuery will match Documents if the relevance rating is above the specified relevance rating MinThreshold and below the relevance rating MaxThreshold. A MinThreshold of 0.0 and a MaxThreshold of 1.0 indicates that the DetectionNeed the RoutingQuery was derived from will be returned from the RetrieveQueries() operation for all matching Documents.

RemoveQuery (QueryCollectionIndex, RoutingQuery)

if *QueryCollectionIndex* includes a query based on the same DetectionNeed as

*RoutingQuery*, that query is removed from the Index

RetrieveQueries (sequence of QueryCollectionIndex, Document, NumberToRetrieve: integer):

DetectionNeedCollection

returns the collection of DetectionNeeds (of maximal length *NumberToRetrieve*) corresponding to those DetectionNeeds which are most closely related to Document Each DetectionNeed in the returned Collection will have a nil RawData property and an Attribute with the name "BaseDocument" and, as value, a DocumentReference identifying the corresponding DetectionNeed in QueryCollectionIndex.

MatchQueryTerms(sequence of QueryCollectionIndex, RoutingQuery, Document)

Document is a Document which was compared to a set of QueryCollectionIndexes in a RetrieveQueries() operation.

sequence of `QueryCollectionIndex` is the set of QCI's used to compare the Document during the `RetrieveQueries()` operation.

`RoutingQuery` is the current `RoutingQuery` representing a `DetectionNeed` returned by the `RetrieveQueries()` operation.

Adds an Annotations to the Document to indicate terms and passages in the Document which match the input `RoutingQuery`.

annotation type `RelevantPassage`

{rank : String}

annotation type `RelevantTerm`

{rank : String}

The query term Annotations indicate where a particular Document matches a `RoutingQuery`. Because these Annotations are specific to an instance of a `RoutingQuery` they are not intended to be a permanent part of the Document. Therefore, users may wish to delete these query term Annotations when they are finished reviewing their `RetrieveQueries` results.

`GetQuery(QueryCollectionIndex, DetectionNeed): RoutingQuery or nil`

`GetQuery` returns the `RoutingQuery` in `QueryCollectionIndex` whose `DetectionNeed` component is identical to the `DetectionNeed` given as an argument. If no such `RoutingQuery` exists in the `QueryCollectionIndex`, `nil` is returned.

### 7.1.5 Query Monitoring

The Monitor object is intended as an advisory object in the Architecture. If no Monitor object is provided, no monitoring or interruption of the `RetrieveDocuments` operation is possible. The `RetrieveDocuments` operation will not fail due solely to the absence of a *nil* Monitor argument.

## Class Monitor

### Description

Tracks the progress of requests to a Detection component. Some programmatic control is provided via `StatusType` and `IntervalType`

### Properties

`StatusType`: one of {`NumDocs` (number of documents processed), `Time` (estimated time to complete - seconds), `Percent` (% of documents processed)}

`IntervalType`: one of {`NumDocs` (number of documents processed), `Time` (interval in seconds between status reports), `Percent` (% of documents processed)}

`Interval`: Integer

`ClientData`: set of string

### Operations

`CreateMonitor (StatusType: one of {NumDocs, Time, Percent}, IntervalType: one of {NumDocs, Time, Percent}, Interval: integer, ClientData: set of string): Monitor`

`StatusType` is the type of report requested. If the type is not supported a reasonable default shall be provided with the type indicated. `IntervalType` indicates the desired type of interval which may differ from `StatusType`. `Interval` indicates the frequency of status information. The `Interval` value behaves according to the `IntervalType`. If `IntervalType` is `Percent` then `Interval = 5` means provide status when each 5% of the documents are processed. `ClientData` is optional user data for the `MonitorProgress` operation

`MonitorProgress (Monitor, DCName: string, Status: integer, MaxStatus: integer, Type: one of {NumDocs, Time, Percent}): Boolean`

`DCName` is the name of the `DocumentCollectionIndex` which is being monitored. `Status` is the current status consistent with type. `MaxStatus` indicates the maximum value `Status` may have for `DCName`. `Type` is the type of progress update provided to the function

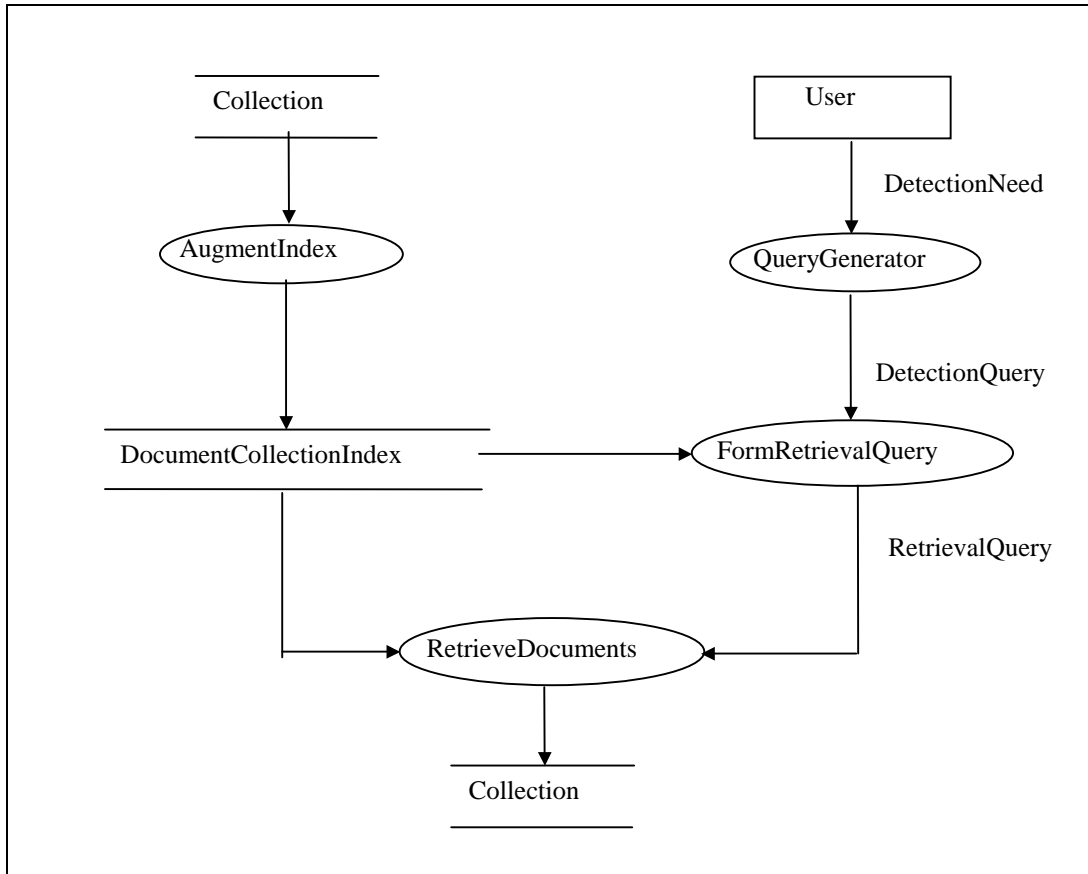
Returns `FALSE` to terminate the search, returns `TRUE` to continue the search

## 7.2 Functional Model

The following functional model diagrams are based on the notation used by Rumbaugh et al. Ovals represent processes (operations); boxes with only a top and bottom represent "data stores" — persistent repositories of data; fully enclosed boxes represent "actors" — active sources of data.

### 7.2.1 Retrospective Retrieval

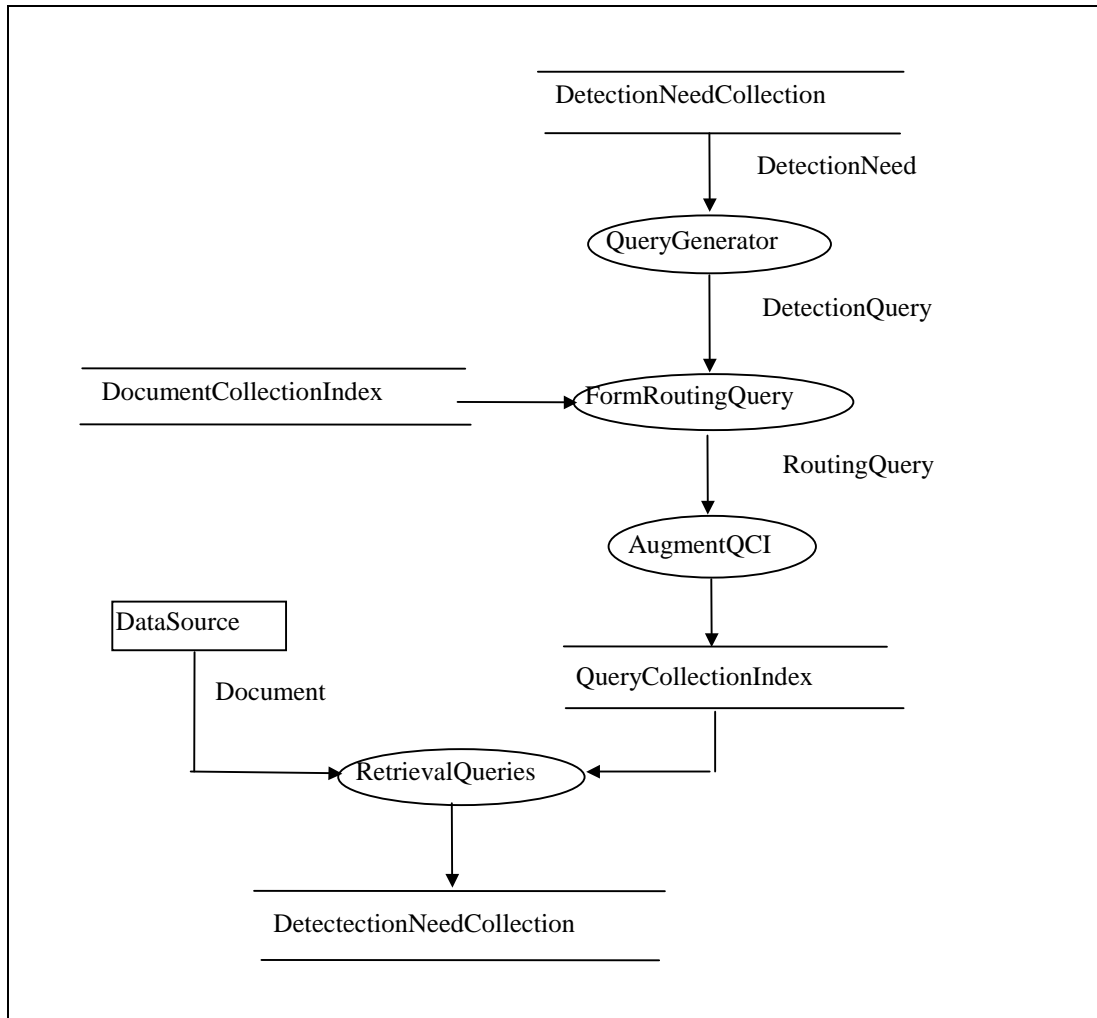
The system begins by converting the DocumentCollection(s) into DocumentCollectionIndex(es), as shown on the left side. To retrieve information from this collection, the User produces a DetectionNeed. This DetectionNeed is converted in two stages, first to a DetectionQuery and then to an RetrievalQuery, as shown in the right column (the latter step may use information, for example, on term weights, from the DocumentCollectionIndex). Finally, the RetrievalQuery is run against the DocumentCollectionIndex to retrieve the documents; this produces a Collection of relevant documents, with a *relevance* attribute for each document.



### 7.2.2 Routing

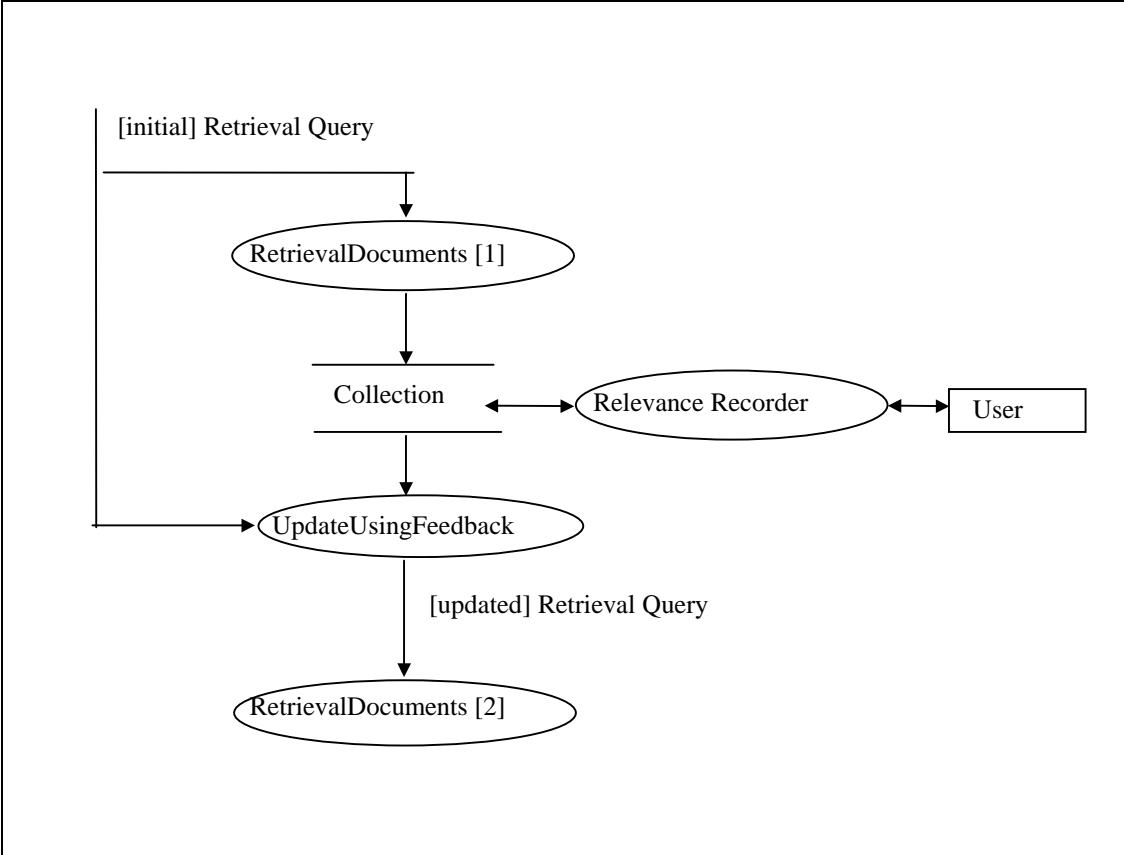
Routing requires a DocumentCollectionIndex which is used to determine weights for the translation of a DetectionQuery into an RoutingQuery. Typically an application will be able to use a pre-existing index (for a Collection of content comparable to the documents to be routed).

Each DetectionNeed (user profile) in the DetectionNeedCollection is translated in two stages: first to a DetectionQuery, and then into a RoutingQuery. These RoutingQueries are then stored and indexed in a QueryCollectionIndex. Finally, this QueryCollectionIndex can be run against a Document to produce a set of relevant queries (profiles), in the form of a DetectionNeedCollection.



### 7.2.3 Relevance Feedback

Relevance feedback begins with an initial RetrievalQuery, which is used to retrieve a set of documents. This operation is shown as "Retrieve Documents [1]" in the figure below (the DocumentCollectionIndex input is not shown), and produces a Collection. A human judge (or possibly an alternative source of relevance judgments, such as an extraction system) then reviews the retrieved documents and records relevance judgments on the Collection using the *relevant* Attribute. This is done using a Relevance Recorder, which is *not* part of the Architecture but would be part of any application system which wished to support relevance feedback. The Collection is then fed, along with the original query, to an UpdateUsingRelevanceFeedback operation, producing an updated query. Finally, the updated query can be used to retrieve a new set of documents (shown as "Retrieve Documents [2]" at the bottom of the figure).





## 8.0 EXTRACTION

Information extraction the extraction from a document of information concerning particular classes of events is a form of document annotation. An extraction engine adds annotations describing the events and their participants. Extraction therefore does not require any operations and classes beyond those already presented. However, because extraction will be a major component of many systems built using the Architecture, this section describes how extraction fits into the current Architecture.

At present the development of extraction engines from a description of a class of events (a "scenario") is a black art practiced by a cadre of information extraction specialists. It is expected that in the future it will be possible for experienced users to customize extraction systems to new scenarios; this would be an interactive process which would draw upon a library of predefined template objects. Appendix A.2 presents the additional object classes which would be needed to support such customization.

### 8.1 Representing Templates as Annotations

In the terminology developed by the Message Understanding Conferences, the information extracted from a document is stored in a (filled) template, which in turn consists of a set of template objects. A template object may contain information about a real-world object (such as a person, product, or organization), a relationship, or an event.

Each such template object provides information about a portion of a document and is therefore represented in the TIPSTER Architecture by an annotation. A particular extraction task will involve several kinds of template objects, for events, people, organizations, etc. Each kind of template object corresponds to a *type* of annotation. Thus the formal specification of a set of template objects corresponds to a set of annotation type declarations. This formal specification is supplemented by a large amount of narrative (the "fill rules") describing the circumstances under which a template object is to be created and the information to be placed in each slot.

Each slot/value pair in the template object is represented as an attribute/value pair on the annotation. Note that the values of attributes can be lists (thus allowing for slots with multiple values) and can be references to other annotations (thus allowing for a hierarchy of filled objects, and allowing for references to other annotations, such as names which have been identified by a prior annotation process). Furthermore, each annotation has a span which can link the object to the text from which it has been derived.

Some applications may want to link an individual slot in the template object to text in the document. This can be done by introducing additional annotations. Instead of having the value of the attribute corresponding to that slot be a string, it would be a reference to an annotation of type *string-annotation*. That annotation would (like all annotations) have a set of spans referencing the text; it would also have a *value* attribute holding the value of the template slot (the "slot filler"). This has been done for one of the slots in the example below, the *role* slot of *personnel*, but could have done it for others.

If an application system involves extractions for multiple scenarios (multiple classes of events), it will be necessary to distinguish the annotations corresponding to different extraction scenarios (so that, for example, one can display all the annotations for one scenario). This can be done by adding a scenario attribute to each annotation. In similar fashion, in an application environment integrating annotation modules from different suppliers, it would be desirable to record the source of particular annotations using an annotator attribute. These additional attributes are not shown in the example below.

### 8.2 An Example

As an illustration of this approach, consider the result of annotating a document consisting of the sentence

The KGB kidnapped ARPA program manager Umay B. Funded.  
with an information extraction system covering terrorist events. The MUC-style template for such an event might look like<sup>7</sup>

```
<EVENT-1> :=
  EVENT_TYPE: KIDNAPPING
  PERPETRATOR: <ORG-1>
  TARGET: <PERSONNEL-1>
```

---

<sup>7</sup> The templates shown here are loosely based on those for the MUC-6 information extraction task.

```

<ORG-1> :=
  ORG_NAME: "KGB"
  ORG_NATIONALITY: USSR
<PERSONNEL-1> :=
  PERSON: <PERSON-1>
  ORGANIZATION: <ORG-2>
  ROLE: "PROGRAM MANAGER"
<ORG-2> :=
  ORG_NAME: ARPA
  ORG_NATIONALITY: USA
<PERSON-1> :=
  PER_NAME: "Umay B. Funded"
  
```

These might be encoded as a set of annotations as follows:

| <i>Text</i>                                                                |                   |            |          |                            |
|----------------------------------------------------------------------------|-------------------|------------|----------|----------------------------|
| The KGB kidnapped ARPA program manager Umay B. Funded.                     |                   |            |          |                            |
| 0...   5...   10..   15..   20..   25..   30..   35..   40..   45..   50.. |                   |            |          |                            |
| <i>Annotations</i>                                                         |                   |            |          |                            |
| Id                                                                         | Type              | Span Start | Span End | Attributes                 |
| 1                                                                          | Name              | 4          | 7        | name_type=organization     |
| 2                                                                          | Name              | 18         | 22       | name_type=organization     |
| 3                                                                          | Name              | 39         | 53       | name_type=organization     |
| 4                                                                          | Event             | 0          | 53       | event_type=kidnapping,     |
| 5                                                                          | Org               | 4          | 7        | perp=[5], target=[6]       |
| 6                                                                          | Personnel         | 18         | 53       | org_name=[1],              |
| 7                                                                          | String-annotation | 23         | 38       | org_nationality=USSR       |
| 8                                                                          | Org               | 18         | 22       | person=[9],                |
| 9                                                                          | Person            | 39         | 53       | organization=[8], role=[7] |
|                                                                            |                   |            |          | value="program manager"    |
|                                                                            |                   |            |          | org_name=[2],              |
|                                                                            |                   |            |          | org_nationality=USA        |
|                                                                            |                   |            |          | per_name=[3]               |

The type declaration package for these annotations is as follows:

```

type package terrorist_event;
annotation type event:
    {event_type: {kidnapping, murder, ...},
     perp: org,
     target: personnel};
annotation type org:
    {org_name: name,
     org_nationality: string};
annotation type personnel:
    {person: person,
     organization: org,
     role: string-annotation};
annotation type person:
    {per_name: name};
annotation type name:
    {name_type: {person, organization, other}};
annotation type string-annotation:
    {value: string};
  
```

## APPENDIX A POSSIBLE EXTENSIONS TO THE ARCHITECTURE

### A.1 Enforcing Type Declarations

In the current Architecture, annotation type declarations serve only as documentation; they are not processed by any component of the Architecture. It may be desirable in future versions of the Architecture to perform type checking based on such declarations. This could involve:

1. creation of a new class of document, **TypeDeclarationDocument**, containing a package of type declarations
2. associating a set of declaration packages with a Collection
3. requiring that any annotation added to a document in a collection conform to the associated type declaration

A number of issues would need to be resolved to implement such a scheme, including the name scoping of annotation types, and the implications of modifying a type declaration after annotations of that type have been created. The overall type checking mechanism would be fairly complex and so has not been included in the current Architecture.

### A.2 Customizable Extraction Systems

The present Architecture treats extraction engines as modules which have been hand-coded for specific tasks (extraction scenarios). In the future, it is expected that there will be more general extraction engines which can be customized *by users* to specific needs. This section considers the additional object classes and data flow which would be entailed.

#### A.2.1 Object Classes

The user would prepare an ExtractionNeed, using a combination of formal specification and narrative description comparable to the "fill rules" for MUC-5. This would then be "translated" to produce a CustomizedExtractionSystem. This translation would be performed by a component which will guide an analyst in producing a CustomizedExtractionSystem; this interactive translation component is labeled *Customize* below. Once a CustomizedExtractionSystem is created, it can be applied to documents in a collection (like other, pre-existing annotators) and will produce templates for the documents.

The Extraction Need would include annotation type declarations for the annotations to be produced. These type definitions will be supplemented by fill rules in the form of *comments*. As the process of translating ExtractionNeeds becomes more formalized, the fill rules will accordingly also become more formalized. For example, the specifications may include the semantic class of particular slot fills. For the present, however, an ExtractionNeed is a kind of TypeDeclarationDocument:

#### Class ExtractionNeed

Type of TypeDeclarationDocument

#### Class CustomizedExtractionSystem

##### Description

a system-specific structure, containing patterns, rules, etc.

##### Operations

Customize (ExtractionNeed): CustomizedExtractionSystem

an interactive process which will guide the user in converting an ExtractionNeed into a CustomizedExtractionSystem.

Extract (which: Collection, destination: Collection, CustomizedExtractionSystem)

the operation which generates templates from documents. Extraction is a special type of annotation, and accordingly the Extract operation is a variant of the Annotate operation (Section 5.6). For each document in collection *which*, if the same document (a document with the same Id) appears in *destination*, annotate that document in collection *destination* with the information extraction templates generated for that document.

## Class Template Object Library

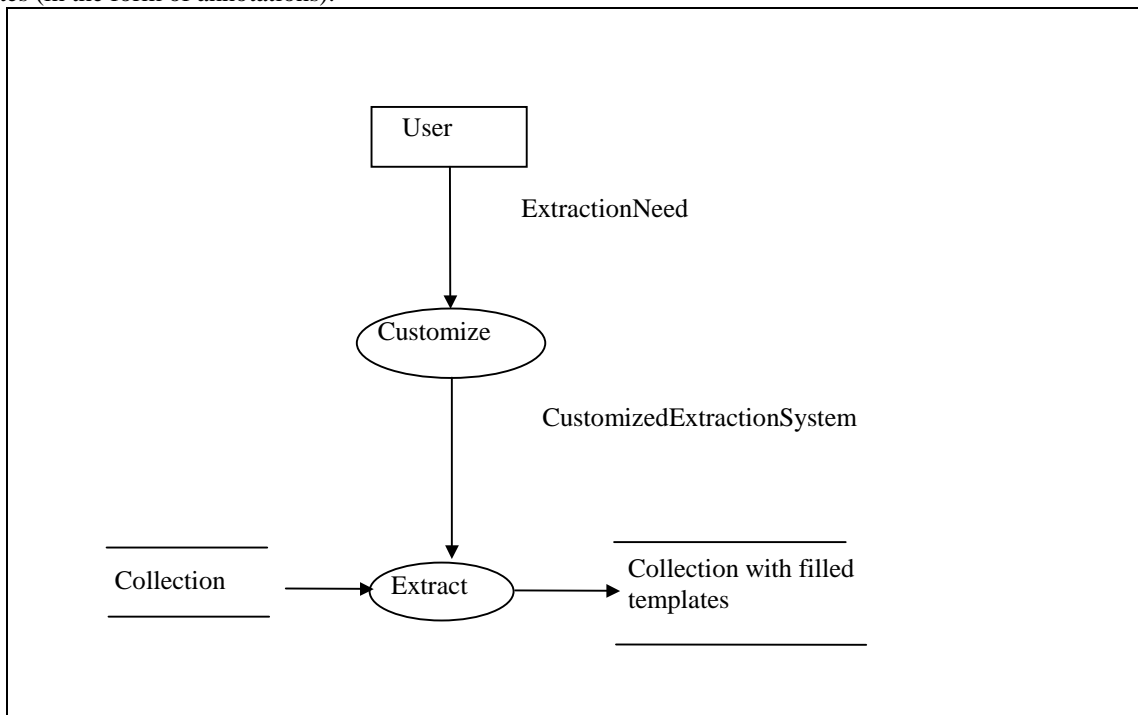
### Description

a set of system-specific rules for extracting various classes of objects, such as persons or organizations; this library could be used in customizing an extraction system to a particular task

### A.2.2 Functional Model

The analyst begins by preparing an ExtractionNeed. The ExtractionNeed would serve as the starting point for customization, which would be performed by the analyst using an interactive customization tool and drawing upon the Template Object Library. The result of this process would be a CustomizedExtractionSystem.

Once a CustomizedExtractionSystem has been created, it can be given a Collection specifying the documents to be annotated (the "*which*" argument) and a Collection where the annotations shall be placed (the "*destination*" argument); it will add to each document of the destination Collection the appropriate templates (in the form of annotations).



## APPENDIX B CLASSES AND THEIR OPERATIONS

This appendix lists all the classes and their operations in an "expanded" form. Instead of showing the properties associated with a class, this appendix explicitly lists the operations to access (and, in some cases, to write) these properties. If property *Comp* of a class is readable, it is accessed by the operation *GetComp*; if it is also writeable, it is set by the operation *SetComp*. In addition, the inheritance of operations from abstract classes has been made explicit: the abstract classes themselves have been removed, and the operations are shown explicitly as part of each class which directly inherited them from the abstract class.

### Class Annotation

#### Operations

CreateAnnotation (Type: string, Spans: sequence of Span, attributes: sequence of Attribute): Annotation  
 GetAttribute (Annotation, name: string): AttributeValue OR *nil*  
 GetAttributes (Annotation): sequence of Attribute  
 GetId (Annotation): string  
 GetSpans (Annotation): sequence of Span  
 GetType (Annotation): string  
 PutAttribute (Annotation, name: string, value: AttributeValue)  
 RemoveAttribute (Annotation, name: string)

### Class AnnotationReference

#### Operations

CreateAnnotationReference (Document, Annotation): AnnotationReference  
 GetAnnotationId (AnnotationReference): string  
 GetCollectionName (AnnotationReference): string  
 GetDocumentId (AnnotationReference): string

### Class AnnotationSet

#### Operations

AddAnnotation (Document, Annotation): string  
 AnnotationsAt (Document OR AnnotationSet, Position: integer): AnnotationSet  
 CreateAnnotationSet (): AnnotationSet  
 DeleteAnnotations (Document OR AnnotationSet, type: string OR *nil*, constraint: sequence of Attribute)  
 GetAnnotation (Document OR AnnotationSet, Id: string): Annotation  
 Length (AnnotationSet): integer  
 MergeAnnotations (AnnotationSet, AnnotationSet): AnnotationSet  
 NextAnnotations (Document OR AnnotationSet, Position: integer): AnnotationSet  
 Nth (AnnotationSet, n: integer): Annotation  
 RemoveAnnotation (Document OR AnnotationSet, Id: string)  
 SelectAnnotations (Document OR AnnotationSet, type: string OR *nil*, constraint: sequence of Attribute): AnnotationSet

### Class Attribute

#### Operations

CreateAttribute (name: string, value: AttributeValue): Attribute  
 CreateAttributeValue (string OR ObjectReference OR sequence of AttributeValue): AttributeValue  
 GetName (Attribute): string  
 GetValue (Attribute): AttributeValue

### Class AttributeReference

#### Operations

CreateAttributeReference (Document, AttributeName: string): AttributeReference

GetAttributeName (AttributeReference): string  
GetCollectionName (AttributeReference): string  
GetDocumentId (AttributeReference): string

### **Class AttributeValue**

#### **Operations**

GetValue (AttributeValue): string OR CollectionReference OR DocumentReference OR  
AttributeReference OR AnnotationReference OR sequence of AttributeValue  
TypeOf (AttributeValue): one of (string, sequence, CollectionReference, DocumentReference,  
AnnotationReference, AttributeReference)

### **Class ByteSequence**

#### **Operations**

ConvertToString (ByteSequence): string  
CreateByteSequence (string): ByteSequence  
Length (ByteSequence): integer

### **Class Collection**

#### **Operations**

AnnotateCollection (which: Collection, destination: Collection, AnnotatorName: string)  
Close (object: Collection)  
CreateCollection (name: string, attributes: sequence of Attribute): Collection  
CreateVolatileCollection (attributes: sequence of Attribute): Collection  
Destroy (name: string)  
FirstDocument (Collection): Document OR *nil*  
GetAttribute (Collection, name: string): AttributeValue OR *nil*  
GetAttributes (Collection): sequence of Attribute  
GetByExternalId (Collection, ExternalId: string): Document OR *nil*  
GetDocument (Collection, Id: string): Document OR *nil*  
GetName (Collection): string  
GetOwner (Collection): string  
Length (Collection): integer  
NextDocument (Collection): Document OR *nil*  
OpenCollection (name: string): Collection  
PutAttribute (Collection, name: string, value: AttributeValue)  
RemoveDocument (Collection, Id: string)  
SetOwner (Collection, string )  
Sync (object: Collection)  
RemoveAttribute (Collection, name: string)

### **Class CollectionReference**

#### **Operations**

CreateCollectionReference (Collection): CollectionReference  
GetCollectionName (CollectionReference): string

### **Class DetectionNeed**

**Type of Document**

#### **Operations**

QueryGenerator (DetectionNeed): DetectionQuery

### **Class DetectionNeedCollection**

**Type of Collection**

### **Class DetectionQuery**

**Type of Document**

#### **Operations**

EditQuery (DetectionQuery)

FormRetrievalQuery (DetectionQuery, sequence of DocumentCollectionIndex): RetrievalQuery  
 FormRoutingQuery (DetectionQuery, sequence of DocumentCollectionIndex): RoutingQuery  
 GetDetectionNeed (DetectionQuery): DocumentReference

## Class Document

### Operations

Annotate (Document, AnnotatorName: string)  
 CopyBareDocument (NewParent: Collection, Document): Document  
 CopyDocument (NewParent: Collection, Document): Document  
 CreateDocument (Parent: Collection, ExternalId: string, RawData: ByteSequence, annotations: AnnotationSet, attributes: sequence of Attribute): Document  
 GetAnnotations (Document): AnnotationSet  
 GetAttribute (Document, name: string): AttributeValue OR *nil*  
 GetAttributes (Document): sequence of Attribute  
 GetExternalId (Document): string  
 GetId (Document): string  
 GetParent (Document): Collection  
 GetRawData (Document): ByteSequence  
 PutAttribute (Document, name: string, value: AttributeValue)  
 ReadSGML (string, Parent: Collection, ExternalId: string): Document  
 SetExternalId (Document, string )  
 WriteSGML (Document, AnnotationSet, AnnotationPrecedence: sequence of string): string  
 RemoveAttribute (Document, name: string)

## Class DocumentCollectionIndex

### Operations

Augment (DocumentCollectionIndex, Collection)  
 Close (object: DocumentCollectionIndex)  
 CreateDocumentCollectionIndex (name: string): DocumentCollectionIndex  
 Destroy (name: string)  
 GetName (DocumentCollectionIndex): string  
 OpenDocumentCollectionIndex (name: string): DocumentCollectionIndex  
 RetrieveDocuments (sequence of DocumentCollectionIndex, RetrievalQuery, NumberToRetrieve: integer, MinThreshold: float, MaxThreshold: float, Monitor or nil): Collection or nil  
 MatchQueryTerms(sequence of DocumentCollectionIndex, RetrievalQuery, Document)  
 Sync (object: DocumentCollectionIndex)

## Class Document Reference

### Operations

CreateDocumentReference (Document): DocumentReference  
 GetCollectionName (DocumentReference): string  
 GetDocumentID(DocumentReference):string

## Class Monitor

### Operations

CreateMonitor (StatusType :one of {NumDocs, Time, Percent}, IntervalType :one of{NumDocs, Time, Percent}, Interval :integer, ClientData :string) :Monitor  
 MonitorProgress (Monitor, DCIName :string, Status: integer, MaxStatus :integer, Type :one of(numDocs, Time, Percent)) :Boolean

## Class QueryCollectionIndex

### Operations

AddQuery (QueryCollectionIndex, RoutingQuery, MinThreshold, MaxThreshold)  
 Close (object: QueryCollectionIndex)  
 CreateQueryCollectionIndex (name: string): QueryCollectionIndex

Destroy (name: string)  
GetName (QueryCollectionIndex): string  
OpenQueryCollectionIndex (name: string): QueryCollectionIndex  
RemoveQuery (QueryCollectionIndex, RoutingQuery)  
RetrieveQueries (sequence of QueryCollectionIndex, Document, NumberToRetrieve: integer):  
    DetectionNeedCollection  
Sync (object: QueryCollectionIndex)  
MatchQueryTerms(sequence of QueryCollectionIndex, RoutingQuery, Document)  
GetQuery(QueryCollectionIndex, DetectionNeed): RoutingQuery or nil

## **Class RetrievalQuery**

### **Operations**

UpdateUsingRelevanceFeedback(RetrievalQuery, relevant\_docs: Collection, sequence of  
    GetDetectionNeed (RetrievalQuery): DocumentReference  
RetrievalQueryFromRelevanceJudgements (relevant\_docs: Collection, sequence of  
    DocumentCollectionIndex, DetectionNeed): RetrievalQuery  
ScoreDocuments (Collection, RetrievalQuery)  
UpdateUsingRelevanceFeedback (RetrievalQuery, relevant\_docs: Collection, sequence of  
    DocumentCollectionIndex): RetrievalQuery

## **Class RoutingQuery**

### **Operations**

GetDetectionNeed (RoutingQuery): DocumentReference  
RoutingQueryFromRelevanceJudgements (relevant\_docs: Collection, sequence of  
    DocumentCollectionIndex, DetectionNeed): RoutingQuery  
UpdateUsingRelevanceFeedback (RoutingQuery, relevant\_docs: Collection, sequence of  
    DocumentCollectionIndex): RoutingQuery

## **Class Span**

### **Operations**

CreateSpan (start: integer, end: integer): Span  
GetEnd (Span): integer  
GetStart (Span): integer

## APPENDIX C C LANGUAGE HEADER FILE

This appendix shows the C language header file corresponding to the classes defined in the TIPSTER Architecture. Each concrete TIPSTER class has been mapped into a C language type of the same name. However, classes, *DetectionNeed*, *DetectionQuery*, and *DetectionNeedCollection*, are not realized as separate C language types. The first two are subtypes of Document in the Architecture, and are treated as instances of the *Document* type; the last is a subtype of Collection, and is treated as a *Collection* in the C language specifications.

Any functional argument which can be of more than one TIPSTER class is declared of type **void\*** in the C-language declarations (however, arguments which can either point to an object of class **X** or be **NULL** are declared as being of class **X**).

Instances of such overloading, and instances where a specific TIPSTER class is represented by a more general C type (e.g., *DetectionNeed* by *Document*) are noted in comments immediately preceding the function type declaration.

```
/* Tipster Architecture header file (tipster.h) */
```

```
typedef char* tip_string;
```

```
typedef int tip_integer;
```

```
typedef float tip_float; typedef int tip_Boolean;
```

```
enum tip_AttributeValueType {STRING, SEQUENCE, COLLECTION_REFERENCE,
    DOCUMENT_REFERENCE, ANNOTATION_REFERENCE, ATTRIBUTE_REFERENCE};
```

```
typedef void* tip_Annotation;
```

```
typedef void* tip_AnnotationReference;
```

```
typedef void* tip_AnnotationSet;
```

```
typedef void* tip_Attribute;
```

```
typedef void* tip_AttributeReference;
```

```
typedef void* tip_AttributeSet;
```

```
typedef void* tip_AttributeValue;
```

```
typedef void* tip_AttributeValueSet;
```

```
typedef void* tip_ByteSequence;
```

```
typedef void* tip_Collection;
```

```
typedef void* tip_CollectionReference;
```

```
typedef void* tip_Document;
```

```
typedef void* tip_DocumentCollectionIndex;
```

```
typedef void* tip_DocumentCollectionIndexSet;
```

```
typedef void* tip_DocumentReference;
```

```
typedef void* tip_Monitor;
```

```
typedef void* tip_QueryCollectionIndex;
```

```
typedef void* tip_QueryCollectionIndexSet;
```

```
typedef void* tip_RetrievalQuery;
```

```
typedef void* tip_RoutingQuery;
```

```
typedef void* tip_Span;
```

```
typedef void* tip_SpanSet;
```

```
typedef void* tip_stringSet;
```

```
void tip_Free(void*);
```

```
tip_string tip_AddAnnotation(tip_Document, tip_Annotation);
```

```
void tip_AddQuery(tip_QueryCollectionIndex, tip_RoutingQuery, tip_float, tip_float);
```

```
void tip_Annotate(tip_Document, tip_string);
```

```
void tip_AnnotateCollection(tip_Collection, tip_Collection,
    tip_string);
```

```
/* Type of argument 1 of AnnotationsAt can be tip_Document or tip_AnnotationSet */
```

```
tip_AnnotationSet tip_AnnotationsAt(void*, tip_integer);

void tip_Augment(tip_DocumentCollectionIndex, tip_Collection);

/* Type of argument 1 can be tip_QueryCollectionIndex or tip_Collection or tip_DocumentCollectionIndex */
void tip_Close(void*);

tip_string tip_ConvertToString(tip_ByteSequence);

tip_Document tip_CopyBareDocument(tip_Collection, tip_Document);

tip_Document tip_CopyDocument(tip_Collection, tip_Document);

tip_Annotation tip_CreateAnnotation(tip_string, tip_SpanSet, tip_AttributeSet);

tip_AnnotationReference tip_CreateAnnotationReference(tip_Document, tip_Annotation);

tip_AnnotationSet tip_CreateAnnotationSet(void);

tip_Attribute tip_CreateAttribute(tip_string, tip_AttributeValue);
/* Type of argument 2 of CreateAttributeValue can be tip_string or tip_CollectionReference or
tip_DocumentReference or tip_AttributeReference or tip_AnnotationReference or tip_AttributeValueSet */
tip_AttributeValue tip_CreateAttributeValue (enum tip_AttributeValueType, void*)

tip_AttributeReference tip_CreateAttributeReference(tip_Document,
    tip_string);

tip_AttributeSet tip_CreateAttributeSet(void);

tip_AttributeValueSet tip_CreateAttributeValueSet(void);

tip_ByteSequence tip_CreateByteSequence(tip_string);

tip_Collection tip_CreateCollection(tip_string, tip_AttributeSet);

tip_CollectionReference tip_CreateCollectionReference(tip_Collection);

tip_Document tip_CreateDocument(tip_Collection, tip_string, tip_ByteSequence, tip_AnnotationSet,
    tip_AttributeSet);

tip_DocumentCollectionIndex
tip_CreateDocumentCollectionIndex(tip_string);

tip_DocumentCollectionIndexSet tip_CreateDocumentCollectionIndexSet(void);

tip_DocumentReference tip_CreateDocumentReference(tip_Document);

tip_QueryCollectionIndex tip_CreateQueryCollectionIndex(tip_string);

tip_QueryCollectionIndexSet tip_CreateQueryCollectionIndexSet(void);

tip_Span tip_CreateSpan(tip_integer, tip_integer);

tip_SpanSet tip_CreateSpanSet(void);
```

```

tip_Collection tip_CreateVolatileCollection(tip_AttributeSet);

tip_stringSet tip_CreatestringSet(void);

/* Type of argument 1 of DeleteAnnotations can be tip_Document or tip_AnnotationSet */
/* Type of argument 2 of DeleteAnnotations can be tip_string or NULL */
void tip_DeleteAnnotations(void*, tip_string, tip_AttributeSet);

void tip_Destroy(tip_string);

/* tip_Document as argument 1 represents Tipster class DetectionQuery*/
void tip_EditQuery(tip_Document);

/* Result of FirstDocument can be tip_Document or NULL */
tip_Document tip_FirstDocument(tip_Collection);

/* tip_Document as argument 1 represents Tipster class DetectionQuery */
tip_RetrievalQuery tip_FormRetrievalQuery(tip_Document, tip_DocumentCollectionIndexSet);

/* tip_Document as argument 1 represents Tipster class DetectionQuery */
tip_RoutingQuery tip_FormRoutingQuery(tip_Document, tip_DocumentCollectionIndexSet);

/* Type of argument 1 of GetAnnotation can be tip_Document or tip_AnnotationSet */
tip_Annotation tip_GetAnnotation(void*, tip_string);

tip_string tip_GetAnnotationId(tip_AnnotationReference);

tip_AnnotationSet tip_GetAnnotations(tip_Document);

/* Result of GetAttribute can be NULL or tip_AttributeValue */
/* Type of argument 1 of GetAttribute can be tip_Annotation or tip_Collection or tip_Document */
tip_AttributeValue tip_GetAttribute(void*, tip_string);

tip_string tip_GetAttributeName(tip_AttributeReference);

/* Type of argument 1 of GetAttributes can be tip_Annotation or tip_Collection or tip_Document */
tip_AttributeSet tip_GetAttributes(void*);

/* Result of GetByExternalId can be tip_Document or NULL */
tip_Document tip_GetByExternalId(tip_Collection, tip_string);

/* Type of argument 1 of GetCollectionName can be tip_AnnotationReference or tip_AttributeReference or
tip_DocumentReference or tip_CollectionReference */
tip_string tip_GetCollectionName(void*);

/* tip_Document as argument 1 represents Tipster class DetectionQuery */
/* Type of argument 1 of GetDetectionNeed can be tip_RoutingQuery or tip_RetrievalQuery or tip_Document*/
tip_DocumentReference tip_GetDetectionNeed(void*);

/* Result of GetDocument can be tip_Document or NULL */
tip_Document tip_GetDocument(tip_Collection, tip_string);

/* Type of argument 1 of GetDocumentId can be tip_DocumentReference or tip_AnnotationReference or
tip_AttributeReference */

```

```
tip_string tip_GetDocumentId(void*);

tip_integer tip_GetEnd(tip_Span);

tip_string tip_GetExternalId(tip_Document);

/* Type of argument 1 of GetId can be tip_Annotation or tip_Document */
tip_string tip_GetId(void*);

/* Type of argument 1 of GetName can be tip_Collection or tip_DocumentCollectionIndex or
tip_QueryCollectionIndex or tip_Attribute */
tip_string tip_GetName(void*);

tip_string tip_GetOwner(tip_Collection);

tip_Collection tip_GetParent(tip_Document);

tip_ByteSequence tip_GetRawData(tip_Document);

tip_SpanSet tip_GetSpans(tip_Annotation);

tip_integer tip_GetStart(tip_Span);

tip_string tip_GetType(tip_Annotation);

/* Result of GetValue can be tip_AttributeReference or tip_AnnotationReference or tip_AttributeValueSet or
tip_AttributeValue or tip_string or tip_CollectionReference or tip_DocumentReference */
/* Type of argument 1 of GetValue can be tip_AttributeValue or tip_Attribute */
void* tip_GetValue(void*);

/* Type of argument 1 of Length can be tip_Collection or tip_SpanSet or tip_DocumentCollectionIndexSet or
tip_AttributeSet or tip_ByteSequence or tip_AnnotationSet or tip_stringSet or tip_AttributeValueSet or
tip_QueryCollectionIndexSet */
tip_integer tip_Length(void*);

tip_AnnotationSet tip_MergeAnnotations(tip_AnnotationSet, tip_AnnotationSet);

/* Type of argument 1 of NextAnnotations can be tip_Document or tip_AnnotationSet */
tip_AnnotationSet tip_NextAnnotations(void*, tip_integer);

/* Result of NextDocument can be tip_Document or NULL */
tip_Document tip_NextDocument(tip_Collection);

/* Result of Nth can be tip_DocumentCollectionIndex or tip_Attribute or tip_Annotation or tip_string or
tip_AttributeValue or tip_QueryCollectionIndex or tip_Span */
/* Type of argument 1 of Nth can be tip_DocumentCollectionIndexSet or tip_AttributeSet or tip_AnnotationSet or
tip_stringSet or tip_AttributeValueSet or tip_QueryCollectionIndexSet or tip_SpanSet */
void* tip_Nth(void*, tip_integer);

tip_Collection tip_OpenCollection(tip_string);

tip_DocumentCollectionIndex tip_OpenDocumentCollectionIndex(tip_string);

tip_QueryCollectionIndex tip_OpenQueryCollectionIndex(tip_string);
```

```

/* Result of Pop can be tip_Attribute or tip_string or tip_AttributeValue or tip_QueryCollectionIndex or tip_Span or
tip_DocumentCollectionIndex */
/* Type of argument 1 of Pop can be tip_AttributeSet or tip_stringSet or tip_AttributeValueSet or
tip_QueryCollectionIndexSet or tip_SpanSet or tip_DocumentCollectionIndexSet */
void* tip_Pop(void*);

/* Type of argument 1 of Push can be tip_AttributeSet or tip_stringSet or tip_AttributeValueSet or
tip_QueryCollectionIndexSet or tip_SpanSet or tip_DocumentCollectionIndexSet */
/* Type of argument 2 of Push can be tip_Attribute or tip_string or tip_AttributeValue or tip_QueryCollectionIndex
or tip_Span or tip_DocumentCollectionIndex */
void tip_Push(void*, void*);

/* Type of argument 1 of PutAttribute can be tip_Annotation or tip_Collection or tip_Document */
void tip_PutAttribute(void*, tip_string, tip_AttributeValue);

/* tip_Document as result represents Tipster class DetectionQuery */
/* tip_Document as argument 1 represents Tipster class DetectionNeed */
tip_Document tip_QueryGenerator(tip_Document);

tip_Document tip_ReadSGML(tip_string, tip_Collection, tip_string);

/*Type of argument 1 of RemoveAttribute can be tip_Collection OR tip_Document OR tip_Annotation */
void tip_RemoveAttribute (void*, tip_string)

/* Type of argument 1 of RemoveAnnotation can be tip_Document or tip_AnnotationSet */
void tip_RemoveAnnotation(void*, tip_string);

void tip_RemoveDocument(tip_Collection, tip_string);

void tip_RemoveQuery(tip_QueryCollectionIndex, tip_RoutingQuery);

/* tip_Document as argument 3 represents Tipster class DetectionNeed */
tip_RetrievalQuery tip_RetrievalQueryFromRelevanceJudgements(tip_Collection, tip_DocumentCollectionIndexSet,
tip_Document);

tip_Collection tip_RetrieveDocuments(tip_DocumentCollectionIndexSet, tip_RetrievalQuery, tip_integer, tip_float,
tip_float, tip_Monitor);
/*argument one is a sequence*/
void MatchQueryTerms(tip_QueryCollectionIndex, tip_RoutingQuery, tip_Document);
/* tip_Collection as result represents Tipster class DetectionNeedCollection */
tip_Collection tip_RetrieveQueries(tip_QueryCollectionIndexSet, tip_Document, tip_integer);

tip_RoutingQuery tip_GetQuery(tip_CollectionIndex, tip_DetectionNeed);

/* tip_Document as argument 3 represents Tipster class DetectionNeed */
tip_RoutingQuery tip_RoutingQueryFromRelevanceJudgements(tip_Collection, tip_DocumentCollectionIndexSet,
tip_Document);

void tip_ScoreDocuments(tip_Collection, tip_RetrievalQuery);

/* Type of argument 1 of SelectAnnotations can be tip_Document or tip_AnnotationSet */
/* Type of argument 2 of SelectAnnotations can be tip_string or NULL */
tip_AnnotationSet tip_SelectAnnotations(void*, tip_string, tip_AttributeSet);

void tip_SetExternalId(tip_Document, tip_string);

```

```
void tip_SetOwner(tip_Collection, tip_string);
```

```
/* Type of argument 1 can be tip_QueryCollectionIndex or tip_Collection or tip_DocumentCollectionIndex */  
void tip_Sync(void*);
```

```
enum tip_AttributeValueType tip_TypeOf(tip_AttributeValue);
```

```
/* Result of UpdateUsingRelevanceFeedback can be tip_RoutingQuery or tip_RetrievalQuery */  
/* Type of argument 1 of UpdateUsingRelevanceFeedback can be tip_RoutingQuery or tip_RetrievalQuery */  
void* tip_UpdateUsingRelevanceFeedback(void*, tip_Collection, tip_DocumentCollectionIndexSet);
```

```
tip_string tip_WriteSGML(tip_Document, tip_AnnotationSet, tip_stringSet);  
/*Arguments 1 & 2 of CreateMonitor can be number of documents, seconds, percent*/  
tip_Monitor tip_CreateMonitor(tip_integer, tip_integer, tip_integer, tip_string);
```

```
/*Argument 4 of MonitorProgress can be number of documents, seconds, percent*/tip_Boolean  
MonitorProgress(tip_Monitor, tip_string, tip_integer, tip_integer, tip_integer);
```

## APPENDIX D PATTERN SPECIFICATION LANGUAGE

Under the current Architecture, Extraction has had minimal discussion and specification. One of the problems that had been identified was the lack of a method to allow developers to express and exchange rules which identify patterns necessary for extraction. An initial solution to this problem has been the design of a Pattern Specification Language. This Language will become part of the Architecture. Pending final material from the TWG for the Pattern Specification Language, a prototype YACC grammar has been placed herein as a placeholder.

```
%{
/* TIPSTER Common Pattern Specification Language Grammar
   Syntax by Jim Cowie, CRL, NMSU
   Semantics and minor mods by Doug Appelt, SRI International
*/
%}

%union {
    Int32 intval ;
    char* string ;
    void* pointer ;
    short* iarray ;
    struct MatchSpec* mspec ;
    struct KleeneOpGroup* kspec ;
    struct RuleRecord* rspec ;
}

%token <string> SYMBOL
%token RIGHT_ARROW
%token DOUBLE_ARROW
%token LEFT_ANGLE_BRA
%token RIGHT_ANGLE_BRA
%token DOUBLE_LEFT_BRA
%token DOUBLE_RIGHT_BRA
%token LEFT_SET_BRA
%token RIGHT_SET_BRA
%token LEFT_BRA
%token RIGHT_BRA
%token LEFT_SQUARE_BRA
%token RIGHT_SQUARE_BRA
%token AMPERSAND
%token BAR
%token <string> NUMBER
%token MINUS
%token <string> QUOTED_STRING
%token ANY
%token TEMP
%token NORM
%token IF
%token THEN
%token ELSE
%token STAR
%token PLUS
%token QUESTION
%token COLON
%token PLUS_COLON
%token SEMICOLON
%token DOUBLE_SEMICOLON
```

```
%token ASSIGN
%token ADD_ASSIGN
%token COMMA
%token STOP
%token EQUAL
%token NOT_EQUAL
%token LESS_OR_EQUAL
%token GT_OR_EQUAL
%token CARAT
%token RULE
%token PHASE
%token PRIORITY
%token TK_INPUT
%token TK_TRUE
%token TK_FALSE
%token ATSIGN

%type <intval> a_c_expression
%type <intval> a_constraint
%type <intval> action
%type <intval> actions
%type <intval> action_exp
%type <intval> anno
%type <intval> anno_type
%type <intval> arbitrary
%type <intval> arglist
%type <intval> assignment
%type <intval> attr_name
%type <intval> basic_pattern_element
%type <mspec> binding
%type <intval> boolean_op
%type <intval> c_expression
%type <intval> constraint
%type <intval> constraint_group
%type <rspec> constraints
%type <intval> decl
%type <intval> declarations
%type <intval> field
%type <intval> function_call
%type <intval> function_name
%type <intval> index
%type <intval> index_expression
%type <intval> index_op
%type <intval> input_token
%type <kspec> kleene_op
%type <intval> macro
%type <intval> macros
%type <intval> macro_header
%type <intval> more_actions
%type <intval> nameddecl
%type <intval> param_list
%type <intval> pattern_element
%type <intval> pattern_elements
%type <intval> phase
%type <intval> post_condition
%type <intval> pre_condition
%type <intval> prioritydecl
%type <rspec> rule
%type <intval> rules
%type <intval> something_or_other
%type <intval> test_op
```

%type <intval> value

%%

phase:

    macros declarations rules

macros:

    | macros macro

macro:

    macro\_header DOUBLE\_ARROW arbitrary RIGHT\_ARROW arbitrary DOUBLE\_SEMICOLON

macro\_header:

    SYMBOL DOUBLE\_LEFT\_BRA param\_list DOUBLE\_RIGHT\_BRA

arbitrary:

    | arbitrary something\_or\_other

something\_or\_other:

    SYMBOL

    | LEFT\_ANGLE\_BRA

    | RIGHT\_ANGLE\_BRA

    | DOUBLE\_LEFT\_BRA

    | DOUBLE\_RIGHT\_BRA

    | LEFT\_SET\_BRA

    | RIGHT\_SET\_BRA

    | LEFT\_BRA

    | RIGHT\_BRA

    | LEFT\_SQUARE\_BRA

    | RIGHT\_SQUARE\_BRA

    | AMPERSAND

    | BAR

    | NUMBER

    | MINUS

    | QUOTED\_STRING

    | ANY

    | TEMP

    | NORM

    | IF

    | THEN

    | ELSE

    | STAR

    | PLUS

    | QUESTION

    | COLON

    | SEMICOLON

    | PLUS\_COLON

    | ASSIGN

    | ADD\_ASSIGN

    | COMMA

    | STOP

    | EQUAL

    | NOT\_EQUAL

    | LESS\_OR\_EQUAL

    | GT\_OR\_EQUAL

    | CARAT

    | TK\_TRUE

    | TK\_FALSE

    | ATSIGN

```
declarations:
    | declarations decl

decl:
    PHASE COLON SYMBOL
    | input_token COLON symbol_list

input_token:
    TK_INPUT

symbol_list:
    SYMBOL
    | symbol_list COMMA SYMBOL

param_list:
    SYMBOL
    | param_list SEMICOLON SYMBOL

rules:
    | rules rule
    | rules error rule

rule:
    namedecl constraints RIGHT_ARROW actions
    | namedecl prioritydecl constraints RIGHT_ARROW actions

namedecl:
    RULE COLON SYMBOL

prioritydecl:
    PRIORITY COLON NUMBER

constraints:
    pre_condition constraint_group post_condition

pre_condition:
    | LEFT_ANGLE_BRA constraint_group RIGHT_ANGLE_BRA

post_condition:
    | LEFT_ANGLE_BRA constraint_group RIGHT_ANGLE_BRA

constraint_group:
    pattern_elements BAR constraint_group
    | pattern_elements

pattern_elements:
    pattern_element
    | pattern_element pattern_elements

pattern_element:
    basic_pattern_element
    | LEFT_BRA constraint_group RIGHT_BRA kleene_op binding
    | LEFT_BRA constraint_group RIGHT_BRA
    | LEFT_BRA constraint_group RIGHT_BRA kleene_op
    | LEFT_BRA constraint_group RIGHT_BRA binding

kleene_op:
```

```

STAR
| PLUS
| QUESTION

binding:
  index_op index

basic_pattern_element:
  LEFT_SET_BRA c_expression RIGHT_SET_BRA
  | QUOTED_STRING
  | SYMBOL
  | function_name LEFT_SQUARE_BRA RIGHT_SQUARE_BRA

c_expression:
  constraint
  | constraint COMMA c_expression

index_op:
  COLON
  | PLUS_COLON

index:
  NUMBER
  | SYMBOL

constraint:
  anno test_op value
  | anno_type

anno:
  anno_type STOP attr_name

anno_type:
  SYMBOL
  | ANY

attr_name:
  SYMBOL

test_op:
  EQUAL
  | NOT_EQUAL
  | LEFT_ANGLE_BRA
  | RIGHT_ANGLE_BRA
  | LESS_OR_EQUAL
  | GT_OR_EQUAL

value:
  NUMBER
  | QUOTED_STRING
  | SYMBOL
  | TK_TRUE
  | TK_FALSE

actions:
  action_exp more_actions
  | action_exp

more_actions:
  COMMA action_exp more_actions

```

```
| COMMA action_exp

action_exp:
  LEFT_BRA IF a_c_expression THEN actions RIGHT_BRA
  | LEFT_BRA IF a_c_expression THEN actions ELSE actions RIGHT_BRA
  | action {$$ = $1 ;}

a_c_expression:
  a_constraint
  | a_c_expression boolean_op a_constraint

boolean_op:
  AMPERSAND
  | BAR

a_constraint:
  index_expression test_op value

action:
  assignment
  | function_call

assignment:
  index_expression ASSIGN ATSIGN
  | index_expression ASSIGN value
  | index_expression ASSIGN index_expression
  | index_expression ASSIGN function_call
  | index_expression ADD_ASSIGN value
  | index_expression ADD_ASSIGN index_expression
  | index_expression ADD_ASSIGN function_call

index_expression:
  COLON index field

field:
  STOP anno_type STOP attr_name
  | STOP anno_type

function_call:
  function_name LEFT_SQUARE_BRA arglist RIGHT_SQUARE_BRA

function_name:
  SYMBOL

arglist:
  index_expression
  | CARAT index_expression
  | value
  | value COMMA arglist
  | index_expression COMMA arglist
  | CARAT index_expression COMMA arglist

%%
```

## APPENDIX E - ALTERNATE DETECTION NEED SPECIFICATION

Detection Need Type 2 (DN2) is an alternate specification to Detection Need specified in 7.1.2. This is the preferred method for new implementations.

A Detection Need Type 2 is an SGML query used for querying a document collection. The Document Type Definition (DTD) is the specification of the grammar of a DN2. The DTD allows an SGML parser/validator to read a DN2 and determine whether it is syntactically correct.

The basic form of a DN2 query is a start tag of <DN2> followed by the various terms that are to be searched for. The DN2 concludes with a end tag of </DN2>. The terms can be straight text surrounded by <TEXT> and </TEXT> tags, or they can be more complex boolean expressions of text terms. The FULL-TERM tag allows the user to combine a text term with some direction about in which field to search for the text. For example, in one query a user could locate all documents that contain "Mark Twain", but using a FULL-TERM tag the user could locate all documents that have "Mark Twain" in the author field.

One note is that it is up to the search engine to document and enable the full capabilities of the DN2. The DTD is simply the specification of format of the DN2. It does not say anything about the semantics of the query.

```
<!ENTITY % yesno "yes|no" >
<!ENTITY % operator "AND|OR|AND-NOT|HEAD-RELATION|OTHER-OPER|INDEPENDENT" >
<!ENTITY % argument "%operator;|TEXT|FULL-TERM" >
<!ENTITY % operarg "SE-APP? & APP-SE? & CONTEXT? & (%argument;)+ " >
<!ENTITY % seapplist "(SE-APP?,APP-SE?)" >
<!ENTITY % saclist "(SE-APP?,APP-SE?,CONTEXT?)" >
<!ELEMENT APP-SE - - CDATA >
<!ATTLIST APP-SE
  NUMBER-TO-RETRIEVE NUMBER          #IMPLIED
  MIN-THRESHOLD    NMTOKEN            #IMPLIED
  MAX-THRESHOLD    NMTOKEN            #IMPLIED
  REFORMULATION    (REFORMULATION | NOREFORMULATION) REFORMULATION
  EXPANSION         (EXPANSION | NOEXPANSION) EXPANSION
  STEMMING          (STEMMING | NOSTEMMING) STEMMING
  HELP              (HELP | NOHELP)    NOHELP
  EXPL              (EXPL | NOEXPL)    NOEXPL
  MATCHES           (MATCHES | NOMATCHES) NOMATCHES
  DOCFREQ           (DOCFREQ | NODOCFREQ) NODOCFREQ
  COLLFREQ          (COLLFREQ | NOCOLLFREQ) NOCOLLFREQ >
<!ELEMENT SE-APP - - (SE-APP-HELP? & SE-APP-EXPL? & SE-APP-DOCFREQ? & SE-APP-
COLLFREQ? &
  SE-APP-INTERNAL-ID? & SE-APP-MATCHES*)>
<!ELEMENT SE-APP-HELP - - CDATA > <!-- Static help string from the search
engine -->
```

```

<!ELEMENT SE-APP-EXPL - - CDATA > <!-- Dynamic explanation from search engine
-->
<!ELEMENT SE-APP-DOCFREQ - - CDATA > <!-- number of document meeting operand -
->
<!ELEMENT SE-APP-COLLFREQ - - CDATA > <!-- number of occurrences of operand -->
<!ELEMENT SE-APP-INTERNAL-ID - - CDATA > <!-- internal id for this node -->
<!ELEMENT SE-APP-MATCHES - - CDATA > <!-- a document id which the meets the
operand -->
<!ATTLIST SE-APP-MATCHES
  SPAN-START NUMBER #REQUIRED
  SPAN-END NUMBER #REQUIRED
  WEIGHT NMTOKEN#REQUIRED>

<!ELEMENT CONTEXT - - (ANN-ATTR) >
<!ATTLIST CONTEXT
  DISTANCE NUMBER #IMPLIED
  ORDERED (ORDERED | UNORDERED) #IMPLIED>

<!ELEMENT FULL-TERM - - (SE-APP? & APP-SE? & TEXT? & ANN-ATTR*) >
<!ATTLIST FULL-TERM
  WEIGHT NMTOKEN #IMPLIED >

<!ELEMENT TEXT - - CDATA -- this will ignore any tags within it -->

<!-- .....Beginning of Operators..... -->

<!ELEMENT INDEPENDENT - - (%operarg;) >
<!ATTLIST INDEPENDENT
  WEIGHT NMTOKEN #IMPLIED >

<!ELEMENT AND - - (%operarg;) >
<!ATTLIST AND
  EXACT (EXACT) #IMPLIED
  FUZZY NMTOKEN #IMPLIED -- fuzzy and exact are mutually exclusive --
  WEIGHT NMTOKEN #IMPLIED >

<!ELEMENT OR - - (%operarg;) >
<!ATTLIST OR
  EXACT (exact) #IMPLIED
  FUZZY NMTOKEN #IMPLIED -- fuzzy and exact are mutually exclusive --
  WEIGHT NMTOKEN #IMPLIED >

<!ELEMENT AND-NOT - - (%operarg;) >
<!ATTLIST AND-NOT
  EXACT (exact) #IMPLIED
  FUZZY NMTOKEN #IMPLIED -- fuzzy and exact are mutually exclusive --
  WEIGHT NMTOKEN #IMPLIED >

<!ELEMENT HEAD-RELATION - - (%operarg;) >
<!ATTLIST HEAD-RELATION
  HR-TYPE (MORPH-VAR|SYNONYM|RELATED|CONSTRAINT|OTHER) #REQUIRED
  WEIGHT NMTOKEN #IMPLIED >

<!-- other search engine defined operator -->
<!ELEMENT OTHER-OPER - - (%operarg; & OTHER-ARGS) >
<!ATTLIST OTHER-OPER
  WEIGHT NMTOKEN #IMPLIED >
<!ELEMENT OTHER-ARGS - - CDATA >

```

```

<!-- .....End of operators..... -->

<!ELEMENT MERGE-INFO - - CDATA >
<!ATTLIST MERGE-INFO
    MERGETYPE (engine-choice) #IMPLIED >

<!-- Special material for INFONEEDs -->

<!ELEMENT INFO-NEED - - (DOC-COLLECTION* & RESTRICT-SET? & FEEDBACK-INFO? &
(%argument;)?)>
<!ATTLIST INFO-NEED
    WEIGHT NMTOKEN #IMPLIED >

<!-- single TIPSTER collection Index -->
<!ELEMENT DOC-COLLECTION - - CDATA >

<!ELEMENT DOCID - - CDATA >

<!ENTITY % b-opernd "B-AND|B-OR|B-AND-NOT|TEXT|FULL-TERM" >

<!ELEMENT B-AND - - ((%b-opernd;),(%b-opernd;)) >
<!ELEMENT B-OR - - ((%b-opernd;),(%b-opernd;)) >
<!ELEMENT B-AND-NOT - - ((%b-opernd;),(%b-opernd;)) >

<!ELEMENT RESTRICT-SET - - ((%b-opernd;)? & DOCID*) >

<!ELEMENT FEEDBACK-INFO - - (DOCID-REL* & TEXT-REL*)>

<!ELEMENT DOCID-REL - - CDATA >
<!ATTLIST DOCID-REL
    REL (REL | NONREL) #IMPLIED
    SPAN-START NUMBER #REQUIRED
    SPAN-END NUMBER #REQUIRED >

<!ELEMENT TEXT-REL - - CDATA >
<!ATTLIST TEXT-REL
    REL (REL | NONREL) REL >

<!-- End of special material for INFONEEDs -->

<!ELEMENT COMMENT - - CDATA -- this will ignore any tags within it -->

<!-- make the attlist actual elements -->
<!ELEMENT ANN-ATTR - - (ANN-TYPE? & ATTR-NAME? & ATTR-TYPE? & ATTR-VALUE?) >
<!ELEMENT ANN-TYPE - - CDATA >
<!ELEMENT ATTR-NAME - - CDATA >
<!ELEMENT ATTR-TYPE - - CDATA >
<!ELEMENT ATTR-VALUE - - CDATA >

<!-- the +(comment) at the end of the DN2 spec is an exception -->

<!ELEMENT DN2 - - (APP-SE? & SE-APP? & MERGE-INFO? & ((%argument;)|INFO-
NEED+)) +(COMMENT) >
<!ATTLIST DN2
    ID ID #IMPLIED
    OUTPUT-QUERY (OUTPUT-QUERY | NO-OUTPUT-QUERY) NO-OUTPUT-QUERY
    OUTPUT-DOCS (OUTPUT-DOCS | NO-OUTPUT-DOCS) OUTPUT-DOCS>

```

