

Multilingual data processing in the CELLAR environment

Gary F. Simons and John V. Thomson
Summer Institute of Linguistics
© 1995 by Summer Institute of Linguistics, Inc.

A paper presented at: Linguistic Databases, 23-24 March 1995, University of Groningen, Centre for Language and Cognition and Centre for Behavioural and Cognitive Neurosciences

Contents:

- 1. The problem of multilingual computing
- 2. An introduction to CELLAR
 - 2.1 The requirements of a computing environment for linguistic research
 - 2.2 A computing environment for linguistic research
 - 2.3 The conceptual modeling language
- 3. Facets of multilingual computing
 - 3.1 System as a repository of multilingual resources
 - 3.2 String as a multilingual object
 - 3.3 Conceptual model as a source of multilingual assumptions
 - 3.4 Attribute as a multilingual alternation
 - 3.5 User as a controller of multilingual preferences
 - 3.6 Interface as a multilingual construction
- 4. Defining multilingual behavior as system resources
- 5. Resources for encoding
 - 5.1 The refined basic model: String and Encoding
 - 5.2 Defining character sets and their characters
 - 5.3 Defining languages and their encodings
 - 5.4 Using encoding information to tokenize strings
- 6. Resources for rendering
- 7. Resources for sorting
- 8. Conclusion
- References

This paper^[fn1] describes a database system developed by the Summer Institute of Linguistics to be truly multilingual. It is named CELLAR--Computing Environment for Linguistics, Literary, and Anthropological Research. After elaborating some of the key problems of multilingual computing (section 1), the paper gives a general introduction to the CELLAR system (section 2). CELLAR's approach to multilingualism is then described in terms of six facets of multilingual computing (section 3). The remaining sections of the paper describe details of how CELLAR supports multilingual data processing by presenting

the conceptual models for the on-line definitions of multilingual resources.

1. The problem of multilingual computing

Ours is a multilingual world and the promise of universal computing cannot be realized until we have an adequate solution to the problem of multilingualism in the computing environment. In the same way that people can be multilingual, our computing systems need to be multilingual. They need the potential to work in any language and to work simultaneously with many languages at the same time.

Many computerists have conceived of the multilingual computing problem as a problem of *special characters*. This approach considers the multilingualism problem to be solved when all the characters needed for writing the languages of interest can be both displayed on the screen and printed. In the early days of computing, when character generation was hard-wired into video terminals and molded directly into print chains and type fingers, this was impossible. Now that graphical user interfaces are the norm, this is not a problem; font-building tools can be used to create customized fonts that hold the needed character shapes.

The writing system is the most visible aspect of language data; that is probably why special characters and fonts have been such a focus of solutions to multilingual computing. But now that graphical user interfaces have made arbitrary fonts commonplace, it is clear that there is much more to multilingual computing than displaying the data. Different languages also have different conventions for many aspects of data processing, such as sorting, keyboarding, spell checking, hyphenation, finding word and sentence breaks, and the like. A truly multilingual computing environment would handle all of these processes in a manner appropriate to the language of the data.

The major software vendors have recognized this problem and, in an effort to find a wider market for their products, have produced versions of their products that are customized for languages other than the one supported by the original version. This is a process generally referred to by the industry as *internationalization* of software. This process confronts many of the problems of multilingual computing, but does not provide a solution. The results are still monolingual (or sometimes bilingual) products which are just as limited as the original monolingual program. But in the same way that people are multilingual, our computing systems need to be multilingual.

We need computers, operating systems, and programs that can potentially work in any language and can simultaneously work with many languages at the same time. A most promising recent development along these lines is the incorporation of the World Script component into version 7.1 of the Macintosh operating system (Ford and Guglielmo 1992). In the late 80s, Apple developed an extension to their font manager called the script manager (Apple 1988). It handles particularly difficult font problems like the huge character inventory of Japanese and the context-sensitive rendering of consonant shapes in Arabic. A script system, in conjunction with a package of "international utilities," is able to handle just about all the language-dependent aspects of data processing mentioned above (Davis 1987). The original script manager's greatest failing was that only one non-Roman

script system could be installed in the operating system. World Script has changed this. It is now possible to install as many script systems as one needs.

World Script begins to make the resources needed for truly multilingual computing available in the operating system, but these resources need to find their way into application software. At this point, programming a new script system or developing an application that takes advantage of the script manager is a low-level job for a skilled system programmer, and progress in this area is slow. What is more, the application environment adds some higher-level requirements for multilingual computing, such as the need for the user to create and manipulate multiple language versions of the same datum, and to have multiple language versions of software interfaces. These additional requirements are developed in section 3 on "Facets of multilingual computing."

The Summer Institute of Linguistics (through its Academic Computing Department) has embarked on a project to build a computing environment that is truly multilingual. The environment is called CELLAR--for Computing Environment for Linguistic, Literary, and Anthropological Research. It is, at the heart, an object-oriented database management system. It includes a high-level programming language that is designed to make it relatively easy to build application programs that are truly multilingual. Before discussing CELLAR's approach to multilingual computing, we begin by giving an introduction to CELLAR as a database system. This is necessary because the multilingual resources of CELLAR are modeled as objects in the database. Thus the definitions of those resources in sections 3 and beyond depend on an understanding of CELLAR's basic object model.

2. An introduction to CELLAR

This introduction to CELLAR proceeds in three parts. Section 2.1 describes the major requirements for a linguistic computing environment which motivated the development of CELLAR. Section 2.2 gives an overview of the major components of the system. Section 2.3 goes into greater depth on the component that will be used throughout the paper to define the formal model of multilingual resources, namely, the conceptual modeling language.

2.1 The requirements of a computing environment for linguistic research

At the heart of computing environment for linguistic research must lie a database of linguistic information. The requirements for such a database system must begin with the nature of the data it seeks to store and manipulate. The following are five of the most fundamental features of the data we work with (both the primary textual data and our analyses of them) and the demands they put on the database:

- The data are multilingual; the database must therefore be able to keep track of what language each datum is in and process each in a manner appropriate to its language.
- The data in a text unfold sequentially; the database must therefore be able to represent text in proper sequence.

- The data are hierarchically structured; the database must therefore be able to express hierarchical structures of arbitrary depth.
- The data elements bear information in many simultaneous dimensions; the database must therefore be able to vest data objects with many simultaneous attributes.
- The data are highly interrelated; the database must therefore be able to encode associative links between related pieces of data.

It is possible to find software systems that meet some of these requirements for data modeling, but we are not aware of any that can meet them all. Some word processors (like Nota Bene, Multilingual Scholar, and those that use the Macintosh World Script system) can deal well with multilingualism (point 1). All word processors deal adequately with sequence (point 2). Some word processors handle arbitrary hierarchy well (point 3), but most do not. The areas of multidimensional data elements and associative linking (points 4 and 5) do not even fall within the purview of word processors. This is where database management systems excel, but they typically do not support multilingualism, sequence, and hierarchy adequately.

The academic community has begun to recognize that SGML (ISO 1986) can be used to model linguistic data; indeed, it is possible to devise data markup schemes that meet all of the above requirements. The Text Encoding Initiative (TEI) guidelines (Sperberg-McQueen and Burnard 1994) give many examples. But SGML-based representations of information are too abstract and too cumbersome for the average researcher to work with directly. This suggests a sixth fundamental requirement for a computing environment that will meet the needs of the linguistic researcher:

- Users need to be able to manipulate data with tools that present conventionally formatted displays of the information; the database must therefore be supported by a complete computing environment that can provide user-friendly tools for creating, viewing, and manipulating information.

Until this requirement is met, it will be difficult for the community of linguistic researchers to realize the promise of the kind of data interchange that the TEI guidelines make possible. See Simons (1994) for a fuller discussion of the role of visual models in a database system.

2.2 A computing environment for linguistic research

CELLAR is a computing environment that is being built expressly to meet the above requirements. See Simons (in press) for a fuller discussion of these requirements and how CELLAR seeks to meet them. At the heart of CELLAR is an object-oriented database and programming system; see Rettig, Simons, and Thomson (1993) for an informal description of its features.

To build an application in CELLAR, one does not write a program in the conventional sense of a structure of imperative commands. Rather one builds a declarative model of the

problem domain. A complete *domain model* contains the following four components:

- *Conceptual model*. Declares all the object classes in the problem domain and their attributes, including integrity constraints on attributes that store values and built-in queries to implement those that compute their values on-the-fly.
- *Visual model*. Declares one or more ways in which objects of each class can be formatted for display to the user.
- *Encoding model*. Declares one or more ways in which objects of each class can be encoded in plain text files so that users can import data from external sources.
- *Manipulation model*. Declares one or more tools which translate the interactive gestures of the user into direct manipulation of objects in the knowledge base.

Because CELLAR is an object-oriented system, every object encapsulates all the knowledge and behavior associated with its class. Thus any object can answer questions, whether from the programmer or the end user, like: "What queries can you answer about yourself?" "What ways do you have of displaying yourself?" "What text encoding formats do you support?" "What tools do you offer for manipulating yourself?" For programmers this greatly enhances the reusability of previously coded classes. For end users this greatly enhances the ability to explore all the capabilities of an application without having to read manuals.

2.3 The conceptual modeling language

Linguists have been using computer systems for decades to model the things in the "real world" which they study. A conceptual model is a formal model in which every entity being modeled in the real world has a transparent and one-to-one correspondence to an object in the model. Relational databases do not have this property; they spread the information about a single entity across multiple tables of a normalized database. Nor do conventional programming languages; even though the *records* of Pascal and the *structures* of C offer a means of storing all the state information for a real world entity in a single data storage object, other aspects of the entity like its behavior and constraints on its relationships to other objects are spread throughout the program.

A conceptual modeling language, like an object-oriented language, encapsulates all of the information about a real world entity (including its behavior) in the object itself. A conceptual modeling language goes beyond a standard object-oriented language (like Smalltalk) by replacing the simple instance variables with attributes that encapsulate integrity constraints and the semantics of relationships to other objects. Because conceptual modeling languages map directly from entities in the real world to objects in the computer-based model, they make it easier to design and implement systems. The resulting systems are easier to use since they are semantically transparent to users who already know the problem domain. See Borgida (1985) for a survey of conceptual modeling languages and a fuller discussion of their features.

A database in CELLAR is defined by a conceptual model. Both when designing conceptual models and when presenting finished ones, we use a graphical notation. When entering the model into CELLAR, we use a formal language. We first present the graphical notation along with an actual example of a conceptual model for a bilingual dictionary. As a conclusion to this section, the syntactic form of the conceptual modeling language is illustrated with the same example.

A key to the graphical notation for conceptual modeling is given in figure 1. The rectangles represent classes of objects. The class name is given at the top of the rectangle; the attributes are listed inside. When the rectangle is dashed, the class is *abstract*. This means that there are no instances of this class; rather, the class is a higher-level node in an is-kind-of hierarchy. All of its subclasses inherit all of its attributes. When the rectangle is solid, the class is *concrete* and it does have instances. Vertical lines link superclasses on top with their subclasses below. The subclass *inherits* all the attributes of the superclass. Thus its full list of attributes consists of the attributes listed for its superclass plus its own attributes.

The *attributes* of a class are listed inside the rectangle. When nothing follows the name of an attribute, its value is a simple string, number, or boolean. Arrows indicate that the attribute value is an instance of another class. A single-headed arrow means that there is only one value; a double-headed arrow indicates that a sequence of values is expected. Solid arrows represent *owning* attributes; these model the part-whole hierarchy of larger objects composed of smaller objects. Dotted arrows represent *reference* attributes; these model the network of relationships that hold between objects. In a CELLAR knowledge base, every object is owned by exactly one object of which it is a part, but it may be referred to by an arbitrary number of other objects to which it is related.

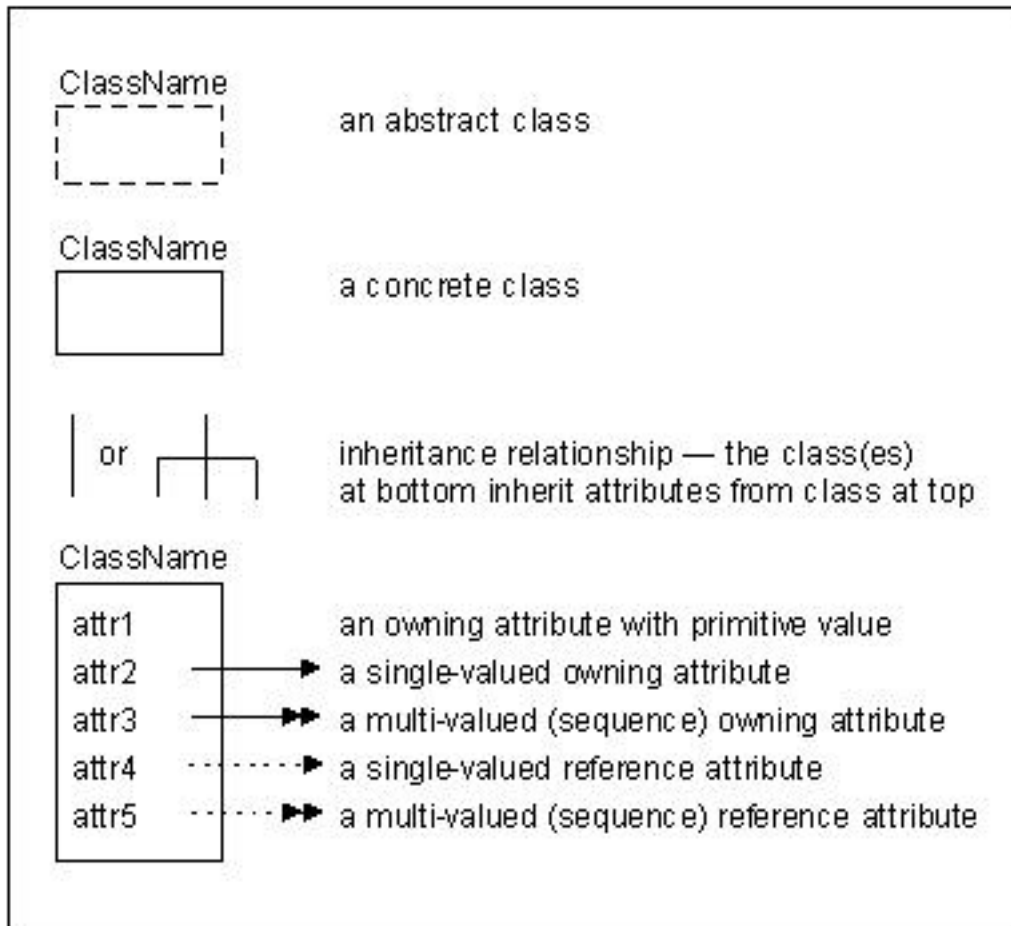


Figure 1: Key to conceptual modeling notation

Figure 2 shows a sample conceptual model for a basic bilingual dictionary. The top-level class in the model is *BilingualDictionary*. It has attributes identifying the two languages, the compiler, and the categories that are allowed for parts of speech and other kinds of information that draw from closed sets of values. The main contents of a *BilingualDictionary* are a sequence of *Entries*, one for each lexical item. The *Entry* has attributes that give the form of the lemma, alternate forms, and the etymology, but the bulk of the information is in a sequence of *Subentries*, one for each sense of meaning for the word. A *Subentry* gives grammatical information as well as definition and cultural notes. Furthermore, it may contain *Examples*, *Idioms*, *morphological Derivations*, and *Xrefs* (semantic cross-references), each of which is modeled as a complex object with further attributes. *Idiom* is a subclass of *Example*; it adds a literal translation to the text and free translation that an *Example* has. The target of a cross-reference (*Xref*) points to the semantically related *Subentry*.

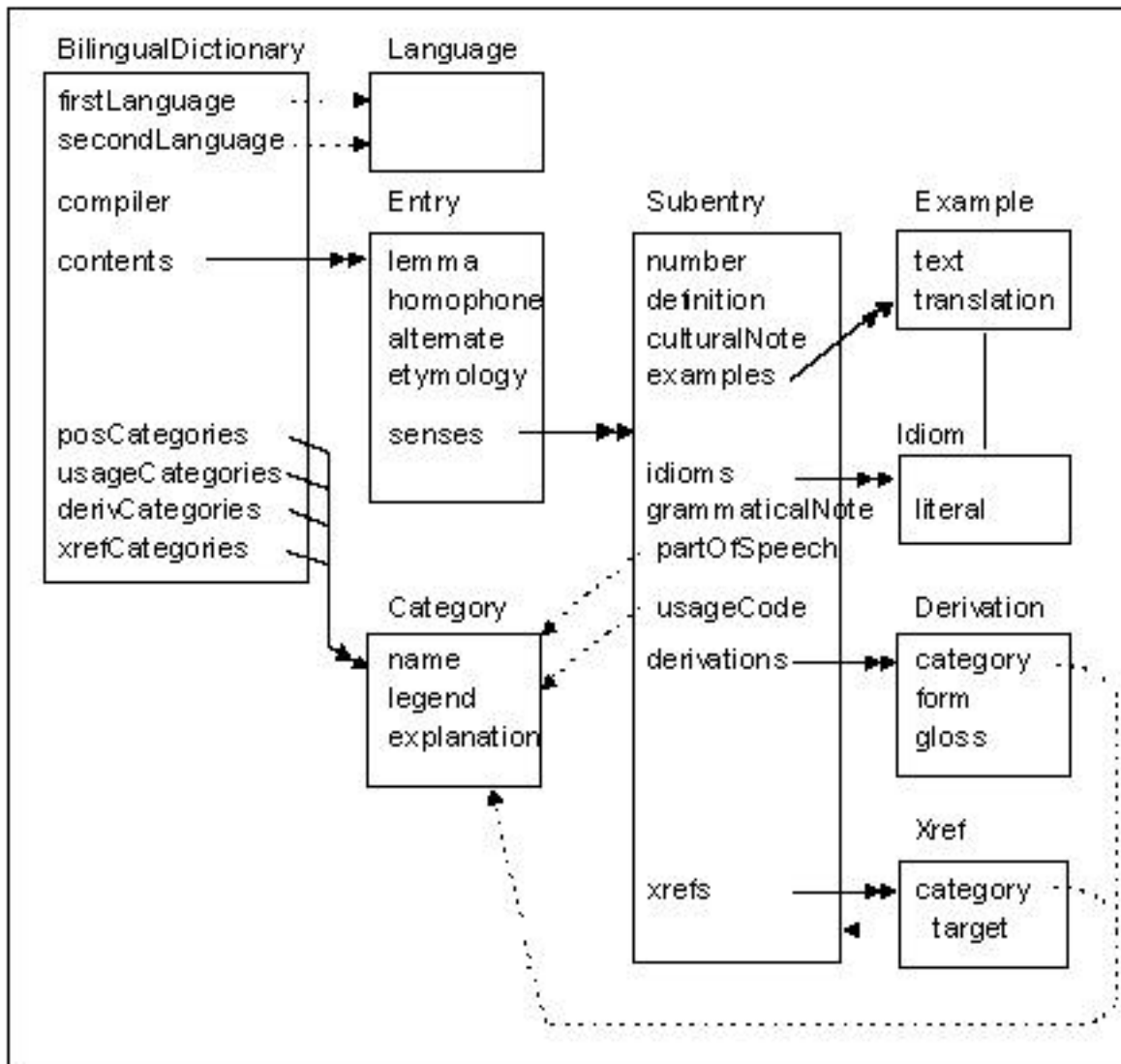


Figure 2: Sample conceptual model of a basic bilingual dictionary

For an illustration of the conceptual modeling language, consider the following definition of the class `Subentry` and some of its attributes:

```
class Subentry has
  definition : String
  examples   : sequence of Example
  partOfSpeech : refers to Category
    in posCategories of owner of my owner
  number     : Integer means
    positionIn("senses", self) of my owner
```

This defines the class `Subentry` to have four attributes. An attribute definition consists minimally of the attribute name and its signature, separated by colon. The signature declares what class of objects are allowed to be attribute values (if the attribute is being set) or are returned as values (if the attribute is being queried). The phrase *sequence of* (or simply *seq of*) indicates that the attribute takes a sequence of values.

The phrase *refers to* indicates that the attribute stores a reference to an object that is part of

some other object. The *in* phrase which follows gives a query that retrieves all the possible target objects in the system. In this case the query is *posCategories of owner of my owner*. *X of Y* means, "Return all the objects generated by getting the X attribute of each of the objects generated by the query Y." *Self* is a query that returns the object itself; *my X* is a shorthand for *X of self*. Thus, *my owner* returns the Entry of which the Subentry is a part; its *owner* is the BilingualDictionary as a whole. The possible values for the *partOfSpeech* of a Subentry are thus any of the Category objects that are stored in the *posCategories* attribute of the BilingualDictionary.

The final attribute in the example is a virtual attribute. It is so called because the value is not really there in the object. It is calculated on-the-fly by executing a query. The *means* keyword indicates that the attribute is virtual. The expression that follows is the query to be executed when the object is asked for the value of the virtual attribute. In this case, the *number* of the sense of meaning is computed by asking the Entry of which this is a sense (that is, *my owner*) to tell us the position in its *senses* attribute of the current Subentry (that is, *self*).

3. Facets of multilingual computing

In our multilingual world, a person may use many languages during the course of a day. When people are multilingual, they do not isolate their use of one language from another; rather, they may mix the languages in a single utterance, or use one language to explain the meaning of an utterance in another. In order to use a computer effectively, these same people need systems that can cope in a similar manner with all the languages they use. For a computing system to be truly multilingual, it needs to cope with multiple languages in the following ways:

- It must be possible for different bits of text to be in different languages.
- It must be possible for a string of text in one language to embed spans of text in other languages.
- It must be possible for the system to make valid assumptions about what language a string is in without requiring a declaration from the user.
- It must be possible for a string of text in one language to be associated with equivalent expressions (that is, translations) in other languages.
- It must be possible for users to control which version they prefer to see when equivalents in many languages are present.
- It must be possible for the interface to a computer system to change in response to the user's language needs and preferences.

The next five subsections discuss these requirements of a multilingual computing system in greater depth and describe how CELLAR meets them.

3.1 System as a repository of multilingual resources

The entry point for CELLAR's model of multilingual computing is the string as an individual data item. String is a fundamental data type in virtually every programming language and database system, and in every system known to us, String is modeled as just a sequence of numerical character codes. But consider the following two propositions which we take to be axiomatic:

- (1) Any bit of textual data stored in a computer is expressed in some language (whether it be natural or artificial).
- (2) Text-related information processing functions (like keyboarding, spell checking, sorting, hyphenation, finding word boundaries, and so on) are language dependent.

For instance, the conventional keyboard layout on English typewriters has QWERTY in the upper left, while the conventional layout for French has AZERTY. The string *noch* would be a misspelled word in English, but is fine in German. In English *ll* sorts between *lk* and *lm*, but in Spanish it sorts between *lz* and *ma*.

These points lead to the following two conclusions:

- (3) A computer system cannot correctly process a bit of textual data unless it knows what language it is in.
- (4) A string, as the fundamental unit of textual data, is not fully specified unless it identifies the language it is in.

CELLAR's model of multilingual computing is based on these two points. Figure 3 illustrates the basic model. Every string of textual data in the user database contains not only its characters but also a tag for the language it is in. These tags take the form of pointers to language definitions that are stored in the system's configuration. These language definitions are objects that declare everything the system knows about how to process data in that language. (The details of these language definitions are explored below in section 4 through section 6.) Thus the computing system is a repository of multilingual resources in two respects: each piece of text is an information resource in some (identified) language, and the definitions of the languages and their processing behavior are key information resources.

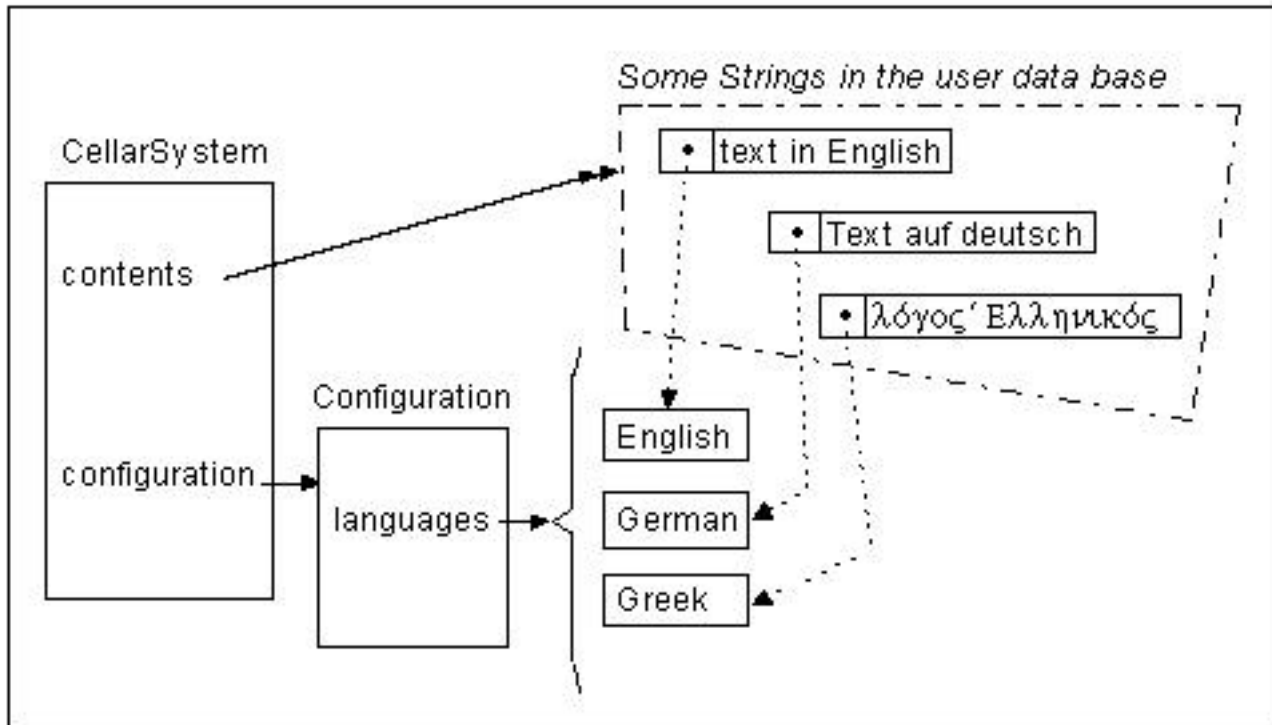


Figure 3: The system as a repository of multilingual resources

3.2 String as a multilingual object

It is not enough that the system as a whole can cope with multiple languages. So must an individual string, since:

- (5) A string of text in one language may include spans of text in another language, and so on recursively.

For instance, a paragraph in English might quote a sentence in German which discusses a Greek word. Such embedding of language within language is a fundamental property of the way humans use language and must be reflected in our computing systems.

CELLAR's conceptual model of String as a data object is therefore a recursive one, in that the value of a string can contain another string. And each of these strings has a built-in *language* attribute to store the pointer to the language it is in. The recursion stops with "primitive strings." These are sequences of character codes with no explicit pointer to language; a primitive string is in the language of the String it is a part of. The value of a string can then be a sequence of a mixture of any number of primitive strings and strings. Thus,

```
class String has
  language : refers to Language
  value    : sequence of PrimitiveString or
String
```

```
class PrimitiveString has
```

value : sequence of CodePoint

Note that `PrimitiveString` is actually a secret of the implementation; the CELLAR user can see and create only `Strings`. But the internal structure of strings is accessible through two built-in attributes of `String`:

`isMonolingual`

Returns *true* if the `String` contains only a `PrimitiveString`; *false* if it contains at least one embedded `String`.

`substrings`

Returns the sequence of `PrimitiveStrings` and `Strings` at the top level of the `String`'s internal structure. The `PrimitiveStrings` are returned as `Strings` with the language explicitly set.

In a typical computer application, mixing of languages is handled by font switches in a flat string. But this flat model is not adequate for it fails to encode the logical structure of the string. Consider the two multilingual strings diagrammed in figure 4. The first might represent a sentence in English which embeds two phrases from Arabic. The second could represent an English sentence that quotes an Arabic sentence that contains an English word. Both multilingual strings, if they are flattened, involve an alternating sequence and English and Arabic primitive strings, but the difference in logical structures is encoded by the different patterns of recursive embedding.

It is essential to encode logical structure explicitly in order to get the correct display of these strings on the screen. In the first case, the left-to-right order of English dominates the whole display with the characters of the embedded Arabic strings being displayed in right-to-left order (as indicated by the reversed order of characters in *2sp* and *4sp*). In the second case, the right-to-left order of Arabic dominates the whole embedded string, so that primitive strings 2 through 4 are displayed in right-to-left order. As a result, the relative positions of strings 2 and 4 are flipped in the two displays. (See Knuth and MacKay 1987 for a complete discussion of text formatting of multilingual documents with mixed direction.) Without support of arbitrary embedding of languages within multilingual strings it is impossible to guarantee that the correct display can be generated when the layout must be changed, such as to accommodate a different column width.

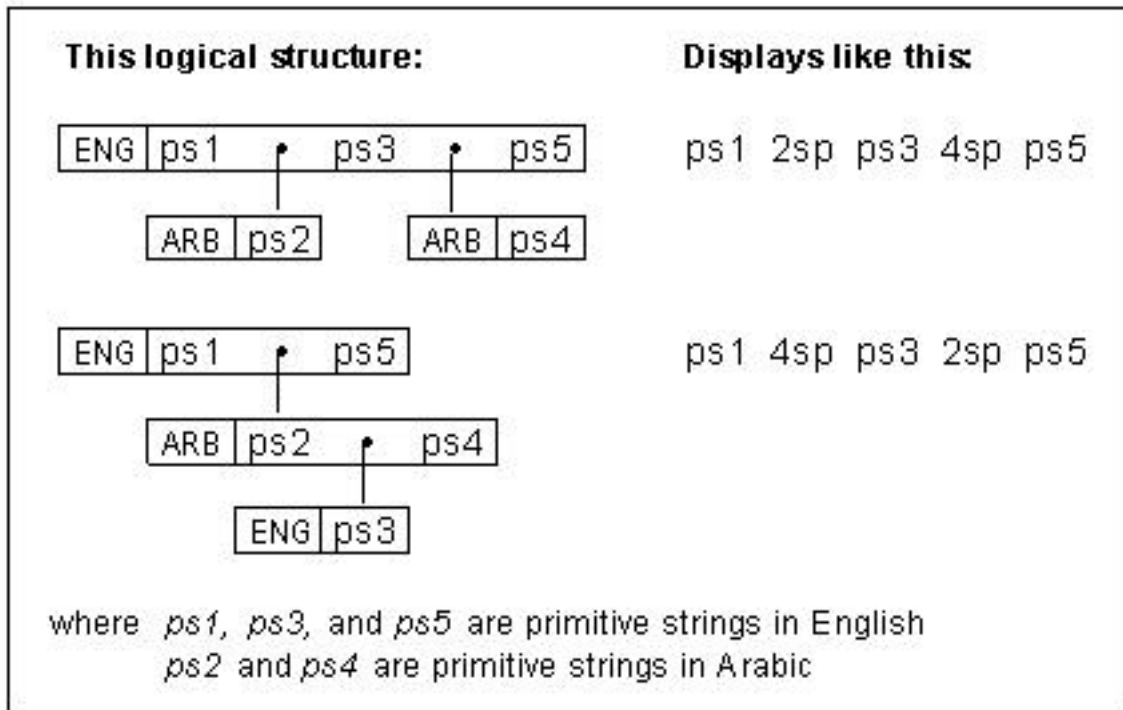


Figure 4: Necessity for embedding in multilingual strings

3.3 Conceptual model as a source of multilingual assumptions

We have proposed that every string in the database should be tagged for what language it is in. To ask the user to explicitly tag each string would be an onerous task; it would not seem at all necessary since:

- (6) The language of a string of text is largely predictable from its context in the database.

For instance, once we know that a complete text is in the Latin language, we can assume that all of its component parts are also in Latin unless we are told otherwise. Once we know that we have a bilingual dictionary of German to French, we can assume that any headword we see is in German and any definition is in French. In the same way, a database implementation of a bilingual dictionary should be able to assume that text entered into the headword field is in the first language and that text entered into the definition field is in the second language. When the two languages involved use completely different writing systems the necessity for automatic switching of language based on context is even more apparent.

CELLAR supports this requirement of multilingual computing by allowing the attribute definitions in a conceptual model to take a *default language* declaration. Indeed, the expectations as to what parts of a data object are in what language are an integral part of its conceptual model. For instance, the bilingual dictionary of section 2.3 (see figure 2) already has attributes at its top-level to record the *firstLanguage* and the *secondLanguage* of the dictionary. The following attribute definitions are then used to tell the system that

definitions are assumed to be in the second language, and that the text of an example is in the first language while the translation is in the second:

```
class Subentry has
  definition : String
  default language is secondLanguage of my dictionary
  examples   : sequence of Example

class Example has
  text       : String
  default language is firstLanguage of my dictionary
  translation : String
  default language is secondLanguage of my
dictionary
```

Note that the identity of the default language is retrieved by a query. It would be possible to express these as literal references to particular languages, but then the conceptual model would not be reusable for dictionaries of other languages. The queries used here assume that a virtual attribute named *dictionary* is implemented; it goes up the ownership hierarchy to find the top-level BilingualDictionary object. It has a very simple recursive definition: on BilingualDictionary it is defined to mean *self*, and on all other objects in the conceptual model it is defined to mean *dictionary of my owner*.

3.4 Attribute as a multilingual alternation

We have already seen that strings may be multilingual by virtue of embedding text in multiple languages. Another dimension of multilingualism for strings is that:

- (7)
- A string of text in one language may have equivalent ways of being expressed (that is, translated) in other languages.

For instance, consider the following situations:

- A programmer writing an application for a multilingual audience wants all the strings of text displayed by the program to have versions in every language of the target audience.
- A linguist writing a dictionary of a vernacular language spoken in an area that has a number of official languages as well as other languages of wider communication wants the definitions and other explanatory notes to be recorded in each of these languages.
- A linguist analyzing a text wants to record the words of the text in three alternate orthographies: a scientific orthography, a traditional orthography, and a proposed new practical orthography. (In our model these are alternate encodings of the string; see section 5.3).

Note the fundamental difference between these needs and those discussed in the previous section. When a string embeds strings in other languages, we are dealing with a single

author's and reader's ability to handle data in multiple languages; the data in different languages are typically meant to be read consecutively. When we deal with multiple translations or encodings of the same data, we are dealing with the possibility that different readers have different abilities to cope with particular languages, and that different authors may be able to write in different languages. A typical reader will use one version or another, not all of them, although sometimes comprehension might be aided by using more than one.

The need to store multilingual alternatives is so fundamental that it is not confined to strings; it could occur at virtually any level in the conceptual model.

(8)

A constructed object expressing information in one language may require alternate constructions to express equivalent information in other languages.

For instance, if a text is modeled as a hierarchical structure of objects rather than being one long string, then translating a paragraph object that is made up of sentence objects may require that a whole new paragraph object be built so that the translated sentences can be reordered into a more appropriate order for the target language.

Due to point (8), we treat the problem of multilingual alternatives as not being confined to the contents of a string, but as being a matter of alternations among related objects. We treat the attribute as the point of alternation; that is, an attribute, even though it is declared to have a single value, is allowed to store language-dependent alternatives for that single value. But not every attribute should allow an alternation; it is a decision the designer of a conceptual model should make. Thus CELLAR provides this capability through the use of the *multilingual* keyword in the signature of an attribute definition. For example, to turn the bilingual dictionary of section 2.3 into a multilingual dictionary in which we did not have to define separate attributes for all the possible defining languages, we would give definitions like the following:

```
class Subentry has
  definition : multilingual String
  default language is secondLanguage of my dictionary
  examples   : sequence of Example

class Example has
  text       : String
  default language is firstLanguage of my dictionary
  translation : multilingual String
  default language is secondLanguage of my
dictionary
```

Note that in Examples, the *text* is still confined to a single language, but that the *translation* could be in many different languages. Note, too, that multilingual alternations do not interfere with the notion of default language. Unless the attribute is told otherwise (such as by the explicit language tagging of the object), an object is assumed to be the alternative for the default language.

3.5 User as a controller of multilingual preferences

Once attributes store multilingual alternatives for the data, it becomes important to determine which alternative to show at any given time. This is not something the conceptual model can determine, for:

- (9) Different users of a computer system have different abilities to read and use different languages.

Thus the fundamental control over which language alternatives should be displayed rests with the user.

CELLAR supports this at the system level by maintaining the user's list of *preferredLanguages* in the Configuration (see figure 5). This is a list of the languages which the user has some competence to read or use. The languages must be chosen from among those that are defined in the system configuration, and they are ranked in order of the user's preference. For example, an English speaker with a good working knowledge of German and a smattering of French might list those three languages in that order. A native speaker of Swahili who understands French well and English less well might list those three.

When the user interface asks an attribute for its value and multiple alternatives are present, the system consults the preferred languages list and returns the first available alternative from the user's preference list. For example, if the preferences list is Swahili, French, English, then if a Swahili alternative is present the system shows that. If not, it tries for a French alternative. Failing that, it tries English, and failing that, it picks one of the available alternatives arbitrarily.

A programmer may want to control the selection of alternatives more precisely in a particular instance. For these purposes, CELLAR provides a refinement to the normal *of* operator for getting the value of an attribute. For instance, whereas *X of Y* is the normal syntax for asking object Y to return its value for attribute X, the query *choose L for X of Y* specifically requests the alternative stored in that attribute for language L. Given that L is a single language and LL is an ordered list of languages, the *choose . . . for* construction has the following variant forms:

choose L for
Return the alternative for language L.

choose LL for
Return the first alternative from the given list.

choose all LL for
Return all alternatives from the given list in the order listed.

choose L ... for
Return the alternative for language L if present; otherwise, the first alternative

according to system preferences.

choose *LL* ... for

Return the first alternative from the given list, or if none present, the first alternative according to system preferences.

choose all ... for

Return all the alternatives that are present in system preference order.

choose all *LL* ... for

As above, but put the alternatives explicitly listed at the front of the result.

L and *LL* above may give literal language identifiers, in which case the programmer has full control. Or, they may be queries that retrieve preferences entered by the user, in which case the user has application-specific control. By using the variants above that return multiple values, the programmer can build user interfaces that display many values for an attribute, not just one. This would be desirable, for instance, for the scenario developed in the preceding subsection where a single dictionary includes definitions in many languages.

3.6 Interface as a multilingual construction

Point (9) above does not apply only to the treatment of data; a multilingual computing environment also needs to be able to provide tools for *manipulating* the data that communicate to different users in the languages they understand best. There are many aspects to this. For example,

- Menu item and button label text may need translation.
- Explanatory text in a dialog may need translation.
- Error messages may need translation.

A typical approach (such as is followed on the Macintosh) is to put all translatable text in a resource file. However, such resource files are only designed to deal with one language at a time. To make a version for another language it is necessary to copy the resource file and translate every text string. The result is a set of monolingual tools.

CELLAR's approach is to provide a truly multilingual tools by using multilingual attributes in the classes that are used to build tools. For example,

- The text of menu items and button labels are multilingual attributes.
- Literal strings placed in windows may be multilingual.
- Exception (error, warning, information) objects use multilingual attributes for their brief message and for their extended explanation.

Simply by manipulating the *preferredLanguages* list in the system configuration, the user can interactively switch the language of the interface. This design with a single multilingual application means that the user will see a mixed language application when only some of the strings are available in the first preference language. For instance, the brief error messages might be translated into a variety of regional languages, while the extended explanations are available in only a few international languages. CELLAR's preference system insures that users will see the available translation that best meets their needs.

But this only solves part of the problem. Building multilingual user interfaces is more than just translating bits of text, because:

(10)

Different languages (due to differences in word order, direction of writing, length of words, and the like) may require different layouts to display equivalent information.

Again, the standard solution is to put layout designs in resource files and to copy them and rearrange them as needed for each new language version.

CELLAR solves this problem, too, in a manner that supports development of a single multilingual application. User interfaces are constructed by nesting a structure of declarative templates, each of which specifies a graphic layout for a set of objects. There are templates to layout horizontal rows, vertical piles, wrapping paragraphs, bordered frames, editable text fields, check boxes, radio buttons, and so on. The problem of alternative layouts for multilingual interfaces is solved by adding another subclass of template: a multilingual template. As a CELLAR object, it has a simple definition:

```
class MultilingualTemplate based on Template has
  template : multilingual Template
```

In other words, `MultilingualTemplate` is a subclass of `Template`. Thus, anywhere it is possible to put a template of any kind, it is also possible to put a `MultilingualTemplate`. The latter has a single multilingual attribute that stores a collection of alternative templates, each of which lays out that substructure of the interface for a different language.

This approach has several advantages over the resource file approach.

- The preference mechanism allows partial translation to work much more easily. For example, a complex system originally created in English might be partly translated into French, and the most frequently-used parts further translated into a local minority language. The preference mechanism allows the minority language speaker to take advantage of whatever is in that language, then whatever is in French, and then see the rest in English. No one has had to construct a special resource file which contains a mixture of the three languages.
- It is not necessary to have a separate program to edit the resources. All the translations are present in the system at once, and can be edited like any other multilingual attributes.

- More information is shared. It is not necessary to make a copy for each language of layouts that do not need to vary.
- Changes can be more incremental. If a different layout is needed for just one part of a dialog, the information about the rest of it can be common.
- All the alternatives can be present together. This makes it much easier to switch the system from one language to another for a user who did not fully understand the explanation in one language and wants to consult a second, or for different users of the same system. All that need be changed is the list of preferred languages.
- The original creator of a tool can give less attention to what will need to be translated, because the language for building user interfaces allows multilingual templates to be inserted at whatever level a cross-language difference is later found.
- CELLAR's window design language goes beyond most resource-based systems in allowing components of a display to be positioned on-the-fly relative to the space they and their neighboring components take up. This greatly reduces the need to build different layouts when the differences between two languages are just a matter of string lengths.

4. Defining multilingual behavior as system resources

The key to CELLAR's ability to function as a multilingual computing environment is that every bit of textual data is explicitly linked to an object that defines its language-specific behavior. The remaining sections of the paper describe these definitions in detail. First, however, it is useful to see an overview.

Figure 5 shows the conceptual model of a CELLAR system and the collection of resources that define its multilingual behavior. The CELLAR system has a number of attributes including:

name

A name to distinguish this CELLAR system from others

contents

The knowledge base of user data objects (including thousands of strings)

domainModels

Domain models (see section 2.2) for all the built-in system classes as well as for user applications

clipboard

The clipboard for copying, cutting, and pasting CELLAR objects

integrityAgenda

A queue of pending object integrity checks and known integrity violations

configuration

A Configuration object which holds the definitions of all multilingual resources installed in the CELLAR system

The system configuration stores the following resources:

languages

Every language known to the system is defined here. Any language can be encoded in more than one way. Thus each Language definition stores a set of LanguageEncodings. See section 5.3 for a detailed discussion.

preferredLanguages

The languages which the user prefers to see when multiple alternatives are available. See section 3.5.

characterSets

Every character set known to the system is defined here. See section 5.2 for a detailed discussion.

defaultEncoding

This pointer selects one of the character sets as the default encoding. If user input or file input ever produces a string of characters with no indication (whether explicitly by tagging or implicitly through the conceptual model) of what its encoding is, it will be set to this by default.

metafonts

These are abstract fonts that are defined across languages, character sets, and hardware platforms. See section 6 for a discussion of how these enhance portability of application code in a multilingual, multiplatform setting.

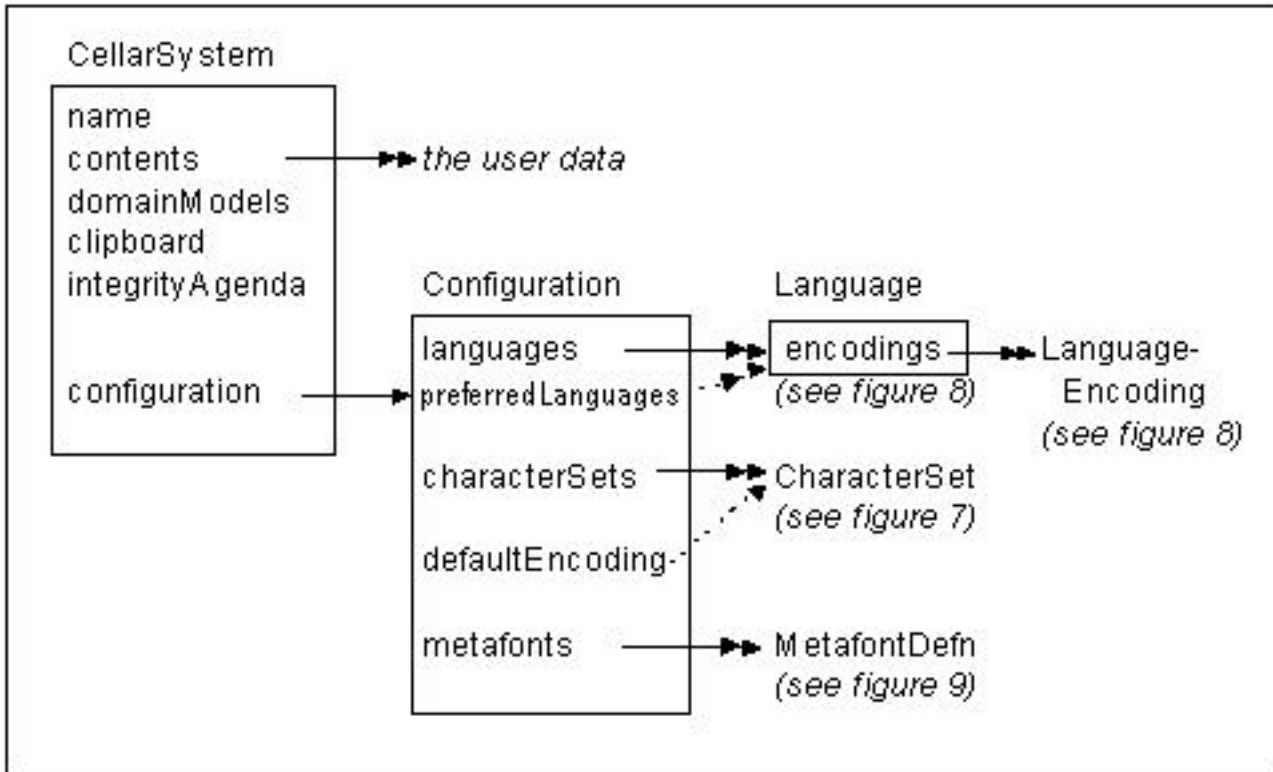


Figure 5: Conceptual model of CellarSystem and Configuration

5. Resources for encoding

In section 3.1 we established that every string in a multilingual computing environment must know what language it is in, but it is actually more complicated than this: every string needs to know both its language and its method of encoding. Section 5.1 explains why. Sections 5.2 and 5.3 define the conceptual models for language and encoding resources. Finally, section 5.4 explains how the encoding resources can be used to automatically tokenize a string (that is, break it into a stream of meaningful subcomponents).

5.1 The refined basic model: String and Encoding

Section 3.2 gives the following conceptual model for String as a basic data type:

```

class String has
    language : refers to Language
    value    : sequence of PrimitiveString or
String

```

But having strings encode their language is not precise enough, for one very basic reason:

(11)

There are many possible ways to map the characters of a particular language onto a set of character codes.

For instance, German can easily be encoded in the ISO 646-021 character set, the ANSI

character set (used with Windows), the Macintosh character set, and the Unicode character set. But the unlauded *u* (,) has a different character code in each case, namely, 126, 252, 159, and 00FC, respectively. It is therefore not enough to know that a particular string is in German; one must know which of the possible encodings it uses. Therefore:

- (12) A string, as the fundamental unit of textual data, is not fully specified unless it identifies the character set by which it is encoded.

At this point one might think that the solution is for the string to identify the character set it is encoded in, rather than the language. But as discussed above under point (2) in section 3.1, two identical strings of characters in the same character set may have different processing requirements if they are in different languages. The string must therefore identify both its language and the way it is encoded by a character set. Thus:

- (13) The information-processing behavior of a string is determined by its method of encoding (which includes its language, its character set, and the way the information units of the language are mapped onto characters).

Therefore, the following statement supercedes points (4) and (12):

- (14) A string, as the fundamental unit of textual data, is not fully specified unless it identifies its method of encoding.

The CELLAR conceptual model of the built-in class String is therefore as follows:

```
class String has
  encoding : refers to Encoding
  value    : sequence of PrimitiveString or
String
```

Encoding is thus the object that encapsulates the knowledge the system has about what the characters in the string represent and how they are to be processed.

The language which a given string represents may not be known. Or, it may be a special-purpose notation that does not warrant formal definition as a language. In these cases, identifying the character set is all that can be said about the string's encoding. When the string is in a known language, the encoding involves the relationship between and language and a character set. The class Encoding is thus defined as an abstract class with two subclasses: CharacterSet and LanguageEncoding.

```
abstract class Encoding has
  name          : String
  description   : String

class CharacterSet based on Encoding has
  characters    : sequence of CharacterDefn
```

```

class LanguageEncoding base on Encoding has
    language      : Language
    characterSet  : refers to CharSet

```

Figure 6 summarizes the basic model in a diagram. It shows that every string stores a pointer to its encoding which may be either a LanguageEncoding or a CharSet. If it is a LanguageEncoding, then that object includes an identification of the language and a pointer to the CharSet in which it is encoded. Note that many Language- Encodings may use (that is, point to) the same CharSet.

5.2 Defining character sets and their characters

It is well known that the number of scripts (or writing systems) in the world is much lower than the number of languages, and that many languages thus use the same script. For instance, most of the languages of western Europe use a Roman script; the Slavic languages of eastern Europe use a Cyrillic script. On the other hand, a single language can be written with more than one script. A classic example is Serbo-Croatian: the Serbs write their language with a Cyrillic script, while the Croats write the same language with a Roman script.

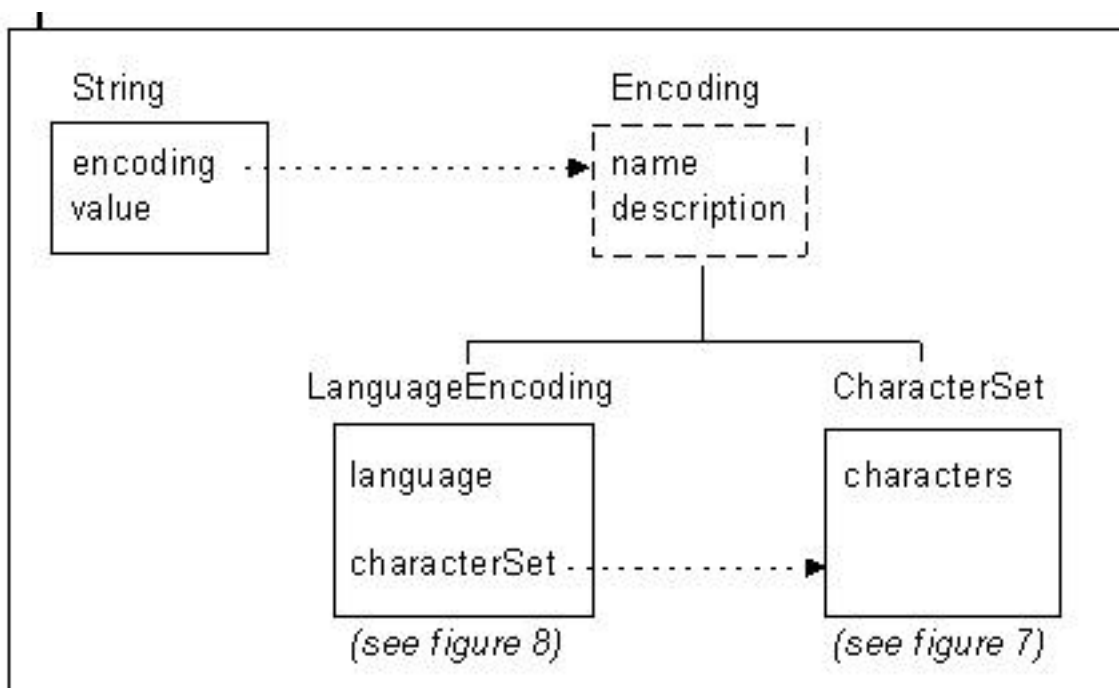


Figure 6: Conceptual model of String and Encoding

Character sets are the computational analog of scripts, and the same relationships to language hold true:

- (15) Many languages may be encoded using the same character set.
- (16) A single language may be encoded using multiple character sets.

In other words, there is a many-to-many relation between languages and character sets. Therefore Language and CharacterSet are modeled independently; the class LanguageEncoding is used to mediate the many-to-many relationship.

The conceptual model of CharacterSet, which is also diagrammed in figure 7, is as follows:

```
class CharacterSet has
  name      : String
  description : String
  codePoints : seq of CodePoint
  baseBeforeDiacritics : Boolean
```

Name is the name by which the character set is known throughout the system. *Description* is documentation for users of the system; it gives a brief explanation of the purpose and source of the character set. *CodePoints* and *baseBeforeDiacritics* are described below.

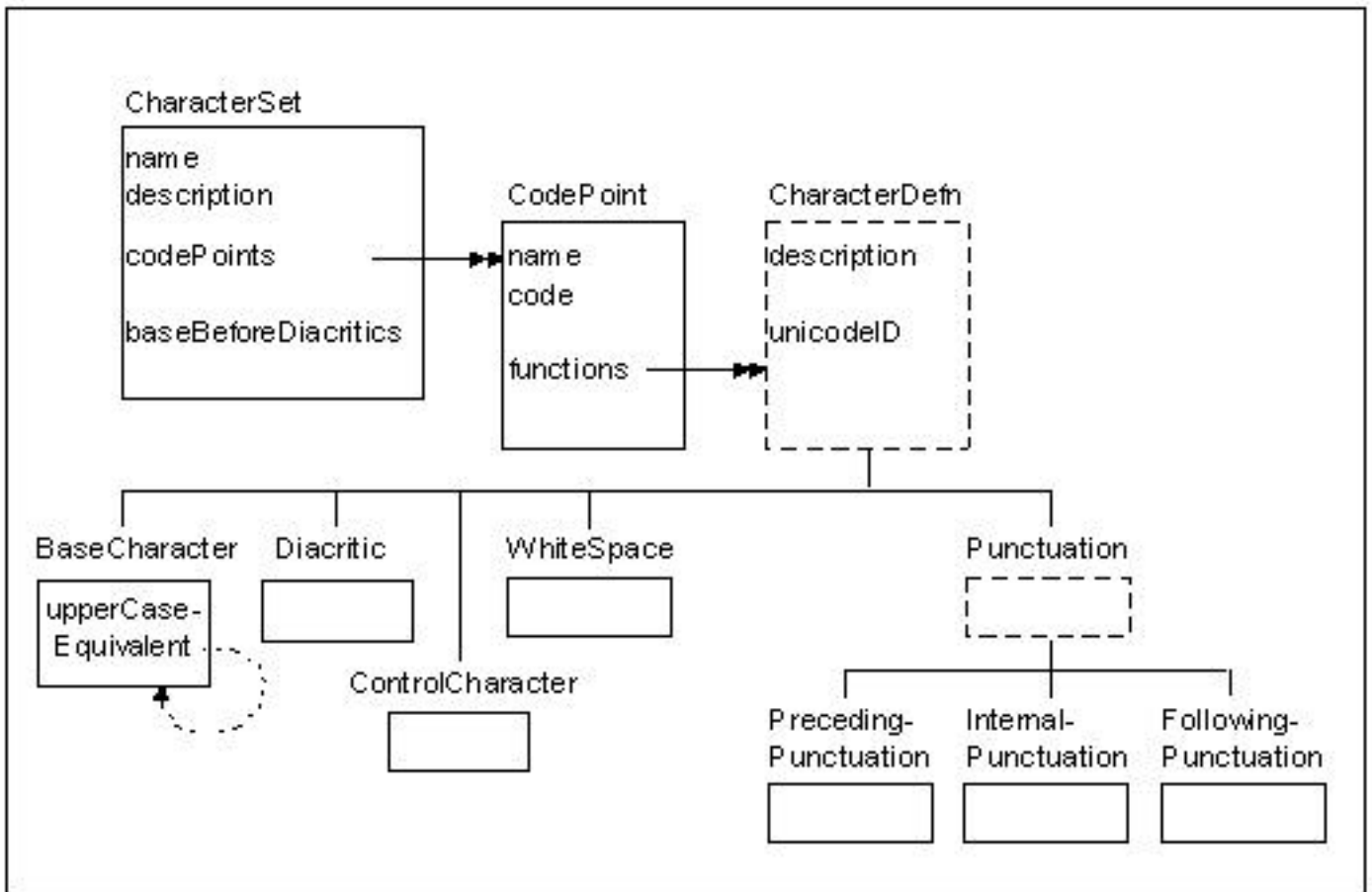


Figure 7: Conceptual model of CharacterSet, CodePoint, and CharacterDefn

Just as a script consists of a collection of graphic symbols (like letters of the alphabet), a character set consists of a collection of characters. But a character is not just a graphic symbol. Rather, it is an abstraction which associates a numerical code stored in the computer with a conventional graphic form. (The actual graphic forms are instantiated by fonts; see section 6.) This association of a numerical code with a graphic form we call a *code point*. But there is more to a character than this alone, because:

(17)

When two languages are encoded with the same character set, it means that characters at the same code point have the same graphic form but not necessarily the same function.

For instance, whereas one language might use the colon (code 58) of the ASCII character set as a punctuation mark, another might use it to signal length of the preceding vowel. Or one language might use the apostrophe (code 39) as a quotation mark while another uses it for the glottal stop consonant. Thus:

(18)

A fully specified character associates a code point (that is, a numerical code and a graphic form) with a function.

Therefore, our design formally distinguishes code points from characters. The *codePoints* attribute of `CharacterSet` is defined to store a sequence of `CodePoints` (see figure 7), and each `CodePoint` in turn stores a sequence of `CharacterDefns`. Each of these character definitions defines an association between its owning code point and the function it represents. When a `LanguageEncoding` declares which characters it uses, it selects these functionally specified `CharacterDefns` rather than `CodePoints`. Even though the `CharacterDefns` are ultimately motivated by the needs of `LanguageEncodings`, they are stored as part of the `CharacterSet` in order to maximize reuse of specifications. For instance, it is not desirable to define a new instance of colon as a punctuation mark for each of the hundreds of languages that can be encoded with a Roman-based character set like ANSI. Rather, the character set owns a single character definition and all language encodings point to the same character definition.

The formal definitions of `CodePoint` and `CharacterDefn` are as follows:

```
class CodePoint has
  name : String
  code : Integer
  functions : seq of CharacterDefn
```

```
abstract class CharacterDefn has
  description : String
  unicodeID : String
```

The *name* is a name for the graphic form associated with the `CodePoint`, such as "apostrophe" or "lowercase alpha." The *description* describes the function of the character if it is other than what would be assumed by the name of the code point. For instance, colon could be described as "vowel length" for the nonstandard use suggested above. Another way to describe the function of a character is to declare the corresponding character in Unicode. For instance, the apostrophe of ASCII (code 39) could be functioning as "apostrophe" (Unicode 02BC), "opening single quotation mark" (Unicode 2018), or "closing single quotation mark" (Unicode 2019). Note that the *functions* of a `CharacterDefn` is a sequence attribute so that such multiple functions for a single code point can be defined.

A third way to specify something about the function of a character is by means of the subclass of `CharacterDefn` that is used. `CharacterDefn` itself is an abstract class; an instance must use one of the subclasses. The subclasses are `BaseCharacter`, `Diacritic`, `ControlCharacter`, `WhiteSpace`, `PrecedingPunctuation`, `InternalPunctuation`, and `FollowingPunctuation`. These specifications allow the system to parse a string of characters into its meaningful parts, like the individual words; more on this below in section 5.4.

Only one subclass adds any attributes, namely, `BaseCharacter`. It adds a pointer to another `BaseCharacter` in the same character set that is the upper case equivalent of this one:

```
class BaseCharacter based on CharacterDefn has
  upperCaseEquivalent : refers to BaseCharacter
    in functions of codePoints of owner of my owner
  lowerCaseEquivalent : seq of BaseCharacter means
    refsFromUpperCaseEquivalentOfBaseCharacter
```

There is no need to set the lower case equivalent. It is computed as a virtual attribute that queries the backreference which is automatically set when *upperCaseEquivalent* is set. It allows multiple values since, for instance, both *a* and *·* might declare *A* as their upper case equivalent.

A final note concerns the *baseBeforeDiacritics* attribute of `CharacterSet`. This stores a Boolean value that declares whether the diacritic characters are encoded before or after their base characters. For instance, if *true*, then *a`e* means *ǣ*; otherwise, it means *aĚ*. By putting this attribute on `CharacterSet` we are saying that even if two character sets have identical code points, there are still different character sets if one encodes diacritics after their bases, while the other puts them before. We consider them different character sets because the encoded strings have different meanings. They are also likely to require different fonts to handle the different discipline for placing the diacritic on its base.

5.3 Defining languages and their encodings

As noted above in point (16), a single language may be encoded with multiple character sets. Figure 8 shows how this is modeled. The `Language` object has an attribute named *encodings* which may store multiple `LanguageEncoding`s. Each `LanguageEncoding` stores a pointer to the `CharacterSet` it uses. The full conceptual model of `Language` is as follows:

```
class Language has
  name : String
  XXXcode : String
  ISOcode : String
  encodings : seq of LanguageEncoding
  preferredEncoding : refers to LanguageEncoding
    in my encodings
```

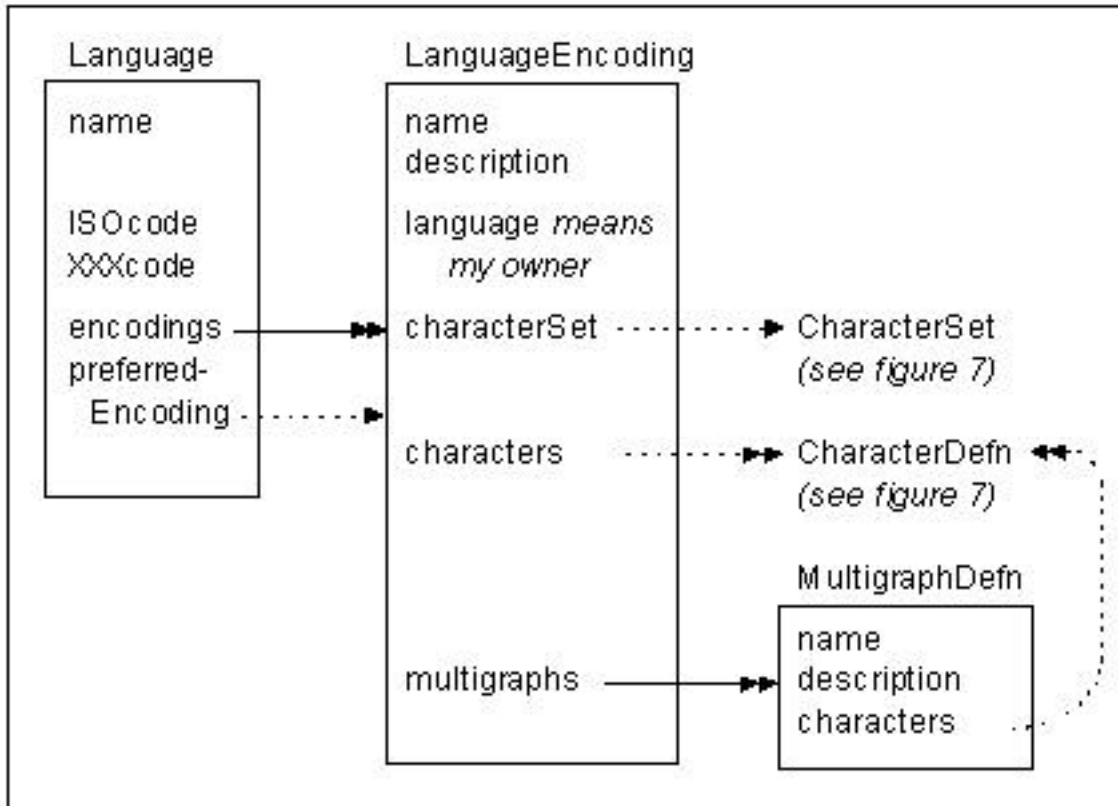


Figure 8: Conceptual model of Language and LanguageEncoding

Name is the display name of the language. *XXXcode* is the three-letter code from the *Ethnologue* (Grimes 1992) which uniquely identifies this language from among the 6,000 plus languages of the world. *ISOcode* is the three-letter code from ISO 639 (ISO 1991); the current committee draft of this standard specifies unique codes for 404 of the world's major languages. *PreferredEncoding* points to one of the encodings for this language; it is the encoding that the system will use if it is asked to create a string in this language and no specific encoding is requested.

The complete conceptual model for LanguageEncoding is as follows:

```
class LanguageEncoding has
  name : String
  description : String
  language : Language means my owner
  characterSet : refers to CharacterSet
    in characterSets of owner of my owner
  characters : refers to seq of CharacterDefn
    in functions of codePoints of my characterSet
  multigraphs : seq of MultigraphDefn
```

The first two attributes give a *name* to the encoding and a brief *description* of its purpose or source. Figure 6 shows that LanguageEncoding has an attribute to specify its *language*. Here we see that this is a virtual attribute; the language is not actually stored here but is retrieved by executing the query: *my owner*. *CharacterSet* indicates which character set is used for this method of encoding the language. It is a pointer to one of the character sets that is defined in the system Configuration (that is, *owner of my owner*; see figure 5).

The *characters* attribute defines the characters that are used in this encoding of the language. It does so by pointing to the appropriate CharacterDefns; note that the *in* clause of the attribute specification limits the selection to only those CharacterDefns that are defined in the character set used by the encoding. By pointing to a selection of the possible CharacterDefns, this attribute does two things. First, it allows the designer of the encoding to omit (and thus disallow) code points that are not used for encoding this language. Second, as noted above in point (17), the same code point can be used for different functions in encoding different languages. This attribute allows the designer of the encoding to select the particular CharacterDefn (and thus the particular function) that is desired for a particular code point.

The final attribute of a LanguageEncoding declares its *multigraphs*. These are sequences of characters which for some purposes (such as sorting or breaking into sound units) should be considered as a single unit; for instance, in German the trigraph *sch* represents a single sound (the grooved alveopalatal fricative) and the digraph *ng* represents another single sound (the velar nasal). A MultigraphDefn has the following conceptual model:

```
class MultigraphDefn has
  name : String
  description : String
  characters : refers to seq of CharacterDefn
    in characters of my owner
```

That is, it has a *name* and a *description* (as would a single character). It also specifies the sequence of *characters* that form the multigraph by pointing to the CharacterDefns for the characters. Note that in this case, the *in* clause is even more restricted than it was for *characters* of the LanguageEncoding. Here the possible values are limited to the characters that are defined in *my owner* (that is, the LanguageEncoding).

5.4 Using encoding information to tokenize strings

One kind of processing that is typically done on text strings is to break them down into meaningful subcomponents, or tokens. For some purposes one might want to view the string as a sequence of characters, for others as a sequence of sound units (where multigraphs are treated as single tokens), and for still others one might want to treat the string as a sequence of wordforms and punctuation marks. The kinds of information stored in CharacterSets and LanguageEncodings makes it possible for CELLAR to perform such tokenization of strings using built- in functions. They are as follows:

`characterTokenize`

Returns a sequence of CharacterDefns for the code points in the String.

`baseCharacterTokenize`

Returns a sequence of Strings, each of which is either a base character with its associated diacritics or a single character of any other type.

`multigraphTokenize`

Returns a sequence of `CharacterDefns` and `MultigraphDefns`.

`wordTokenize`

Returns a sequence of `Strings` (for spans of base characters and diacritics) and Punctuation characters. `WhiteSpace` and `ControlCharacters` are dropped out.

`languageTokenize`

Returns a sequence of monolingual `Strings`.

Each of these is implemented as a built-in virtual attribute of class `String`. For instance, *characterTokenize* of "abc" would return a sequence of three `CharacterDefns`.

Even the simplest tokenizer, *characterTokenize*, makes use of what is known about the encoding of the string. Suppose that a language encoding for English uses the undifferentiated double-quote character (" , ASCII code 34) to mark both opening and closing quotes. The language encoding thus includes two `CharacterDefns` for code point 34, one as `PrecedingPunctuation` and the other as `FollowingPunctuation`. CELLAR uses a simple built-in algorithm that considers position relative to white space and the ends of the string to determine the function of each occurrence of an ambiguous code point. This disambiguated tokenization could be used, for instance, to convert the string to another character set in which different code points and glyphs are used for opening and closing quotes.

More sophisticated tokenizers allow grouping of characters that function together as a single unit at some level. *BaseCharacterTokenize* allows one to identify the base characters of a string and determine which diacritics go with each. CELLAR can do this easily based on its knowledge of which characters are base characters, which are diacritics, which are neither, and whether diacritics attach to the preceding or following base character. *MultigraphTokenize* makes use of CELLAR's knowledge of multigraphs in the string's `LanguageEncoding` to divide a string into a sequence of single characters and multigraphs.

One of the most useful tokenizers is *wordTokenize*. It divides strings into wordforms (made up of successions of base characters and diacritics) and punctuation characters. *LanguageTokenize* takes apart a multilingual string and returns the sequence of monolingual strings it contains. This differs from *substrings* (section 3.2) in that the latter takes apart only the top layer of structure in a multilingual string, while *languageTokenize* completely flattens the structure converting all layers to a single sequential stream.

6. Resources for rendering

Following Becker (1984), we refer to the process of presenting encoded data for display to users in its proper graphic form as *rendering*. In CELLAR, as in most graphical user interface software, we depend on the fonts installed in the operating system to render strings. However, any given font is designed to instantiate the visual forms of the code points in a particular character set. Therefore:

It is not meaningful to display a given string with a particular font unless that font instantiates the character set in which the string is encoded.

The basic model of CELLAR, in which each string knows how it is encoded, makes it easy to ensure that a string is never displayed with a font that does not match its encoding. We need only be sure that each font is associated with the character set it is designed to display, and then ensure that the character set of the font matches the character set of the string when a request is made to display the given string with a particular font.

Since each font is designed for one character set, we store each font definition in the *fonts* attribute of the CharacterSet it goes with (see figure 9). While matching fonts to character sets handles the most basic requirement of the rendering component of a multilingual computing environment, we still fall far short of achieving adequacy. Three major problems remain, all of which relate to portability and reusability of program code.

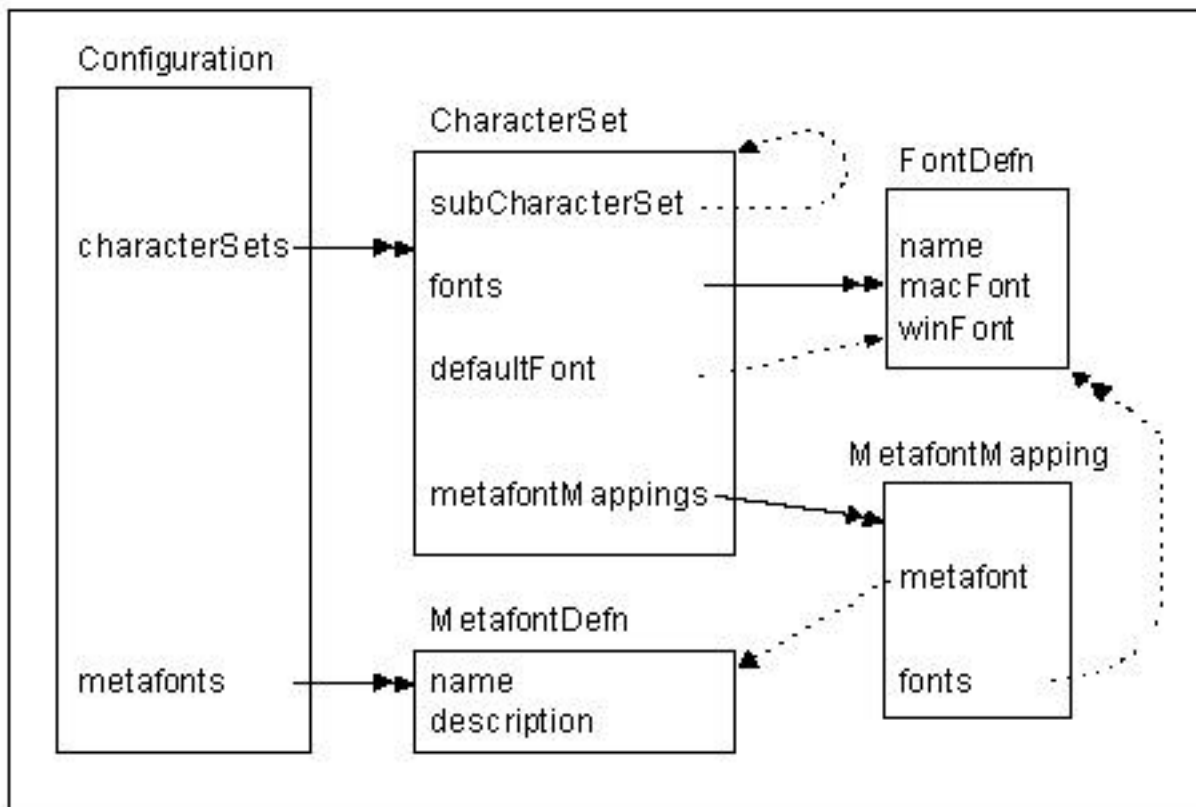


Figure 9: Conceptual model of fonts and metafons

The first problem deals with moving a program that specifies fonts from one hardware platform to another:

(20)

Two different operating systems may have different names for what is essentially the same font.

In such a case, a program that works on one platform may cease to work on the other without being changed. Conversely:

(21)

Two different operating systems may have the same name for two different fonts.

In this case, the program will work on the second platform without crashing but the results will be different. In fact, point (19) above argues that those results are wrong. For these reasons, CELLAR programs do not select fonts by naming the operating system's font directly. Rather, fonts are selected indirectly by means of a `FontDefn` object. It associates a name for CELLAR programs to use with the exact names of the actual fonts on the host system. The conceptual model for `FontDefn` is as follows:

```
class FontDefn has
  name      : String
  macFont   : String
             default is my name
  winFont   : String
             default is my name
```

MacFont and *winFont* give the name of the actual font on a Macintosh system and on a Windows system, respectively. These are the only platforms we support at present. For each new platform that is supported, an attribute must be added. For the sake of convenience, these system-specific names default to the name of the CELLAR font. An explicit `nil` value must be set to indicate that the font is not supported on a particular platform.

The second problem relates to the reusability of display code in a multilingual environment, namely:

(22)

A truly multilingual program cannot anticipate what languages and character sets will occur in the data; thus it cannot be written using character-set-specific font names.

CELLAR therefore supports the notion of a metafont. The conceptual model for the definition is simply the following:

```
class MetafontDefn has
  name      : String
  description : String
```

The *name* is used like a font name when selecting fonts in a program. *Description* is brief documentation that describes the purpose of the metafont. For instance, the default CELLAR installation defines five metafonts:

System

For menu items, button labels, and other controls

DialogData

For data fields in dialog boxes

Fixed

A fixed pitch font for use in displays of data

Serif

A serif font for use in displays of data

SansSerif

A sans serif font for use in displays of data

Programmers thus write dialogs and data displays in terms of these font names with confidence that the system will select an appropriate actual font for each string that is presented for display at run time.

The connection between metafonts and actual fonts are specified in `MetafontMappings` that are part of the definition of each `CharacterSet` (see figure 9). The conceptual model is as follows:

```
class MetafontMapping
  metafont : refers to MetafontDefn
             in metafonts of owner of my owner
  fonts : refers to seq of FontDefn
           in fonts of my owner
```

The mapping associates one of the metafonts defined in the `Configuration` (*owner of my owner*) with one or more fonts defined for this `CharacterSet` (*my owner*).

When the system is asked to display a given string in a particular metafont, it does the following: (1) locates the character set in which the string is encoded, (2) searches its metafont mappings to find the mapping for the requested metafont, and (3) selects the first font listed in the mapping which is found to be installed in the host operating system. If the character set has no mapping for the metafont, or if none of the mapped fonts are present, then the default font for the character set is used. Note that it is a necessary part of adding a new character set to the system to define at least one font for it to serve as the default.

This definition solves a third problem of portability. Namely:

(23)

A program cannot anticipate what actual fonts will be installed on a computer system that will some day run it.

Quite apart from the problems of cross-platform portability, there is even a problem of portability from one installation to another on the same platform. The use a default font for each character set, and of a list of fonts in preference order for each `MetafontMapping`, means that programs can be written without worrying about what fonts will actually be installed on the host system.

The implication from the fact that `CharacterSets` own `FontDefns`--that a given font is valid for displaying strings in only one character set--is not exactly right. This is because:

(24)

If a font is appropriate for displaying a string encoded with a given character set, it is also appropriate for displaying strings which use a character set whose code points form a proper subset of the original character set.

For instance, the ASCII character set is a subset of the ANSI character set. A font designed for the ASCII character set does not supply enough graphic forms to display all strings encoded in ANSI. However, a font for the ANSI character set does supply a compatible graphic form for every code point in the ASCII character set, and therefore could be appropriately used with any ASCII string. Therefore, the conceptual model of `CharacterSet` adds the following attribute:

```
enrich CharacterSet with
  subCharacterSet : refers to CharacterSet
    in characterSets of my owner
```

We can then use the automatic backreference to define the inverse attribute:

```
enrich CharacterSet with
  superCharacterSets : refers to seq of CharacterSet
    means refsFromSubCharacterSetOfCharacterSet
```

This now makes it possible to formally define the notion of "What fonts are available for displaying this string?" (There is also a parallel definition for available metafonts.) We embody this query in a virtual attribute on `String` called *availableFonts*. That is,

```
enrich String with
  availableFonts : seq of FontDefn
    means availableFonts of my encoding
```

The string finds the answer by asking its encoding. If the encoding is a `LanguageEncoding`, it needs to in turn ask the character set. Thus,

```
enrich LanguageEncoding with
  availableFonts : seq of FontDefn
    means availableFonts of my characterSet
```

We are now asking the `CharacterSet` what fonts are defined for it. The fonts available for a given `CharacterSet` are those that it defines directly in its *fonts* attribute, plus the fonts that are available to its *superCharacterSets*. (The square brackets signify building a stream of objects which concatenates the stream generated by the first embedded query with that generated by the second.) That is:

```
enrich CharacterSet with
  availableFonts : seq of FontDefn
    means [ my fonts,
           availableFonts of my superCharacterSets
        ]
```

Note that this definition follows the chain of supersets recursively. Recursion terminates when *my superCharacterSets* returns no value.

7. Resources for sorting

The placement of sorting resources in the CELLAR system is based on the following two observations:

- (25) Different languages, even though they may use the same character set, have different conventions for sorting.
- (26) A single language may have multiple sorting conventions; a researcher may invent new sorting sequences for analytical purposes.

From point (25) we conclude that collating sequences belong in the system as a component of `LanguageEncoding`. When strings that are encoded directly in a `CharacterSet` (with no intervening `Language- Encoding`) are compared, the sequence implied by the numerical codes for their code points is used; in other words, the user cannot control the sort order unless the string is in a language. From point (26) we conclude that each `LanguageEncoding` needs to be able to store a number of possible collating sequences. Thus the *collatingSequences* attribute is defined to hold multiple values. Another attribute, *preferredCollatingSequence*, is needed to point to the collating sequence that should be used if no particular sequence is requested. Figure 10 diagrams these extensions to the model of `LanguageEncoding`.

CELLAR's approach to defining arbitrary collating sequences for sorting is inspired by the approach used to specify a string comparison function in the Macintosh Script Manager (Apple 1993:B-38). The core idea is that sorting is conceptually a three-phase process, although in practice the phases are interleaved so that all three can be stopped as soon as a difference between the strings is found. (Actually there is a fourth phase, not important conceptually, which skips any initial text that is identical in both strings. This is supported by a customized routine called *init*.)

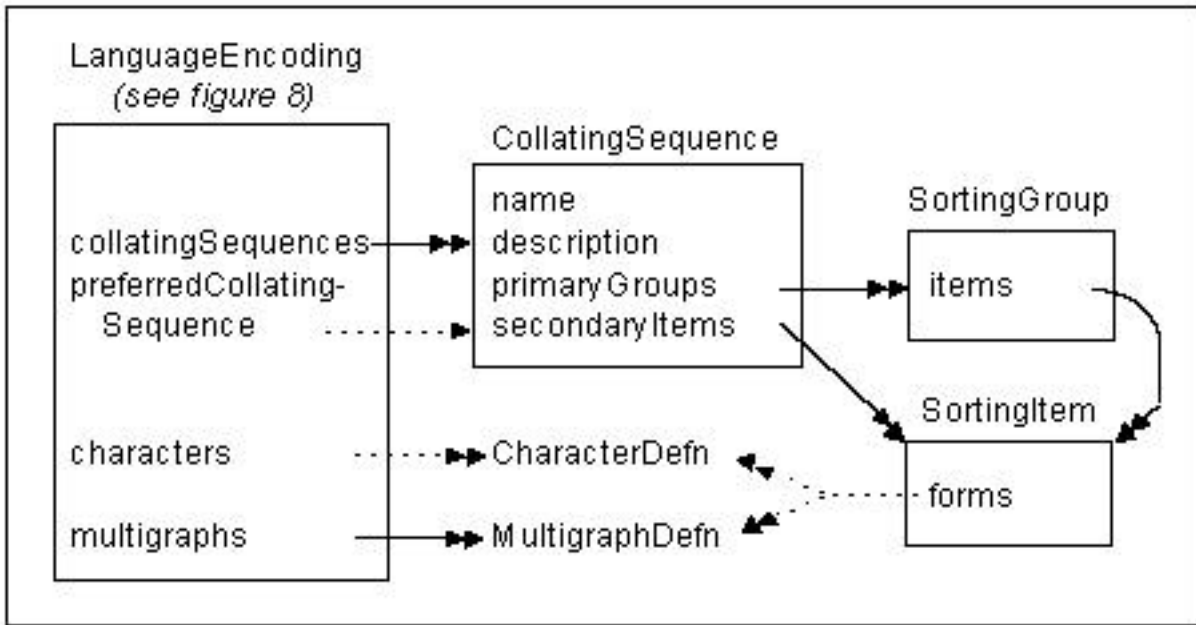


Figure 10: Conceptual model of CollatingSequence

Phase 1 (called the *fetch* routine) divides the string into sorting units. These are typically characters, but some collating sequences treat groups of letters as if they were a single one; for instance, *ll* in Spanish is treated as a single sorting unit that comes between *l* and *m*. Some characters (such as white space and punctuation) may be discarded altogether in this phase.

Phase 2 (called the *project* routine) assigns a primary sort position to each sorting unit, then compares corresponding sorting units of the two strings. The first pair of corresponding units that has a different primary sort position determines the relative sort position of the two strings. This handles the core of sorting: things like putting *a* before *b*. It can also handle treating *a* as equivalent to *A* if these two sorting units are assigned to the same primary sort position.

Phase 3 (called the *vernier* routine) occurs only if phase 2 fails to find any difference between the two strings. It assigns a secondary sort position to each unit, and compares corresponding sorting units. The first pair of corresponding units that has a different secondary sort position determines the relative sort position of the two strings. For example, if two words are otherwise identical, we might wish to sort those that are capitalized before those that are not. To achieve this, upper and lower case versions of the same letter are given the same position in the primary sequence, but different positions in the secondary sequence.

The Script Manager requires procedural programming to define these three phases for a new collating sequence. A person who wants a customized sorting sequence must write small programs to "fetch" a sorting unit from a string, "project" a sorting unit onto a primary sequence number, and a "vernier" routine to make the secondary distinctions between units that are the same in primary sequence.

CELLAR's main contribution in the area of sorting is based on the assertion that:

(27)

The three-phase sort process can be easily specified by a nonprogrammer, and still efficiently implemented, using a declarative table-based approach.

CELLAR's collating sequence tables are basically made up of sorting units grouped by primary position, with the order within each group indicating the secondary ordering. This provides the information needed for all three phases:

Phase 1

We assume that at the start of any string, the longest sequence of characters which corresponds to one of the defined sorting units is the next sorting unit. (This assumption is needed because, for example, *l* and *ll* may both be sorting units. If we find *ll*, we have to assume it is that unit, not two *l* units.)

Phase 2

Primary sort sequence is determined by the position which the sorting unit's group occupies in the main sequence.

Phase 3

Secondary sort sequence is determined by the position of the sorting unit within its group.

To handle cases where two character sequences are to be treated as absolutely identical, CELLAR allows a sorting unit to have more than one concrete representation. This allows us, for example, to make a collating sequence where upper case letters are treated as exactly identical to their lower case equivalents.

It is useful to extend this in one more way. To see why, consider the problem of handling tone marks. Suppose we represent rising tone using a caret at the end of the syllable, while syllables that don't have rising tone are unmarked. Then, suppose we want to sort words as if tone were not marked, except that if words are otherwise identical, we want to sort rising tone after unmarked tone. For example, *ab[^]ec* comes after *abec*, but both are between *abeb* and *abed*.

If we put caret anywhere in the primary sequence, then the system will compare it with the following character (the *e* in the example above) from the words that don't have the caret. Depending on where we put it, *ab[^]ec* will come either after or before all the other three words. If we leave it out, the system has no idea how to order *ab[^]ec* relative to *abec*. Because the caret comes at the end of the syllable, it can follow any letter of the alphabet, so it is extremely clumsy to try to solve the problem by entering each combination with caret into the primary sequence as a digraph.

What we need is to be able to compare a character, for secondary sequencing purposes, with the *absence* of that character in the other string. To allow this, CELLAR's collating sequences have a secondary-only sequence of sorting units, one of which may be an empty

string. If the system finds one of these units in one string and a primary unit in the other, it behaves as if it had instead found an empty string in the one that had the primary unit. The primary unit is reserved to be compared with the next primary unit in the first string, and the difference between the two secondary-only units (one empty) will determine the sort sequence if it depends on the secondary sequence and there is no earlier difference.

CELLAR's actual conceptual model for a collating sequence (see figure 10) is as follows:

```
class CollatingSequence has
  name      : String
  description : String
  primaryGroups : seq of SortingGroup
  secondaryItems : seq of SortingItem.

class SortingGroup has
  items : seq of SortingItem

class SortingItem has
  forms : refers to seq of CharacterDefn or Multigraph
  in [characters of my owner, multigraphs of my
owner]
```

Here are some examples of problems which this approach to sorting can solve:

Capitalized words are sorted as if uncapitalized unless strings are otherwise equal, in which case the capitalized word comes first. This is achieved by putting each character in its own sorting item, but putting corresponding upper and lower case sorting items in that order into the same group.

Capitalization is completely ignored. This is achieved by putting upper and lower case equivalents into the same sorting item.

Accents affect only secondary ordering; each accented vowel is represented by a single code point. Each accented vowel is a sorting item; each is placed in a sorting group with the unaccented vowel that has the same base vowel.

Accents affect only secondary ordering; an accent is represented by a separate diacritic character following the vowel. This can be done two ways. Either do it like the preceding case, noting each vowel-diacritic combination as a multigraph, and then treating each multigraph as a sorting item. Or, treat all the diacritic characters as sorting items in the secondary sequence.

Accented vowels, in either of the above representations, have a distinct spot in the primary sequence. The only difference is that each accented vowel is in a sorting group by itself.

Digraphs have a distinct spot in the primary sequence. Each digraph is noted as a multigraph and treated as a sorting item in its own group. These are then handled automatically as if they were a single character.

Certain characters are ignored. A completely ignorable character is simply left out of the

collating sequence. If it is ignorable only for primary ordering, it is placed in the secondary sequence.

A particularly challenging problem in conventional systems is what to do when comparing strings that are multilingual. The Macintosh approach leaves it entirely up to the application program to keep track of where a string changes language and to figure out what to do about it. CELLAR, by contrast, has built-in knowledge of where strings change language. Therefore, it is possible to compare strings which have sections in different languages.

The basic approach is to allow the language of each monolingual substring to divide that substring into sorting units. Then, as we come to compare pairs of sorting units, if they are in the same language we naturally let that language determine the sort sequence. If they are not in the same language, we can't meaningfully compare the units. Currently, we therefore simply compare the names of the languages. This means, for example, that any character in English comes before any character in French.

Other approaches are possible. It may be sensible, for example, to interfile strings that are encoded in the same character set, even though they encode different languages. For this purpose we could define a collating sequence for a character set that would allow us to compare sorting units that originated from strings of two different languages. A further refinement in this case would be to treat the language identity as a tertiary sorting sequence consulted only if the characters were otherwise the same.

8. Conclusion

All of the features of multilingual computing that are described above have been implemented in the CELLAR system. But much more can be done. Hyphenation, spell checking, and language-specific treatment of numbers and dates are some features we have not begun to work on. In progress is the specification of a finite-state transduction component that can be used to implement language-specific mapping from keyboard to code points, transliteration from one encoding of a language to another, and context-sensitive rendering from character to font glyph (Simons 1989).

These are some of the details of multilingual data processing that can be filled in now that we have built a framework for multilingual computing. Indeed, we feel that this framework is the most significant contribution of our work, for the processing details alone cannot give us computing systems that are truly multilingual--they must be part of an environment that addresses all the facets of multilingual computing.

Footnote 1: We are indebted to many for making the research reported herein possible. We have collaborated (Simons as project director and Thomson as lead programmer) in the design of CELLAR ever since the project began in 1989. The following team of Smalltalk programmers (listed with length of service) have worked to implement these designs: Larry Waswick (5 years), Sharon Correll (4 years), Marc Rettig (2 years), Jim Heimbach (2 years), Nathan Miles (1 year), and David Harrison (6 months). Three other

colleagues--Jonathan Kew, Randy Regnier, and John Wimbish--have contributed materially to the development of our conceptual model for multilingual computing as they have attempted to use CELLAR to build applications. The project has been funded in large part through a grant from Wycliffe Bible Translators (Huntington Beach, CA).

References

- Apple Computer. 1988. The script manager. *Inside Macintosh* 5:293-322. Reading, MA: Addison-Wesley.
- _____. 1993. International resources. Appendix B of *Inside Macintosh: Text*. Reading, MA: Addison-Wesley.
- Becker, Joseph D. 1984. Multilingual word processing. *Scientific American* 251(1):96-107.
- Borgida, Alexander. 1985. Features of languages for the development of information systems at the conceptual level. *IEEE Software* 2(1): 63-72.
- Davis, Mark E. 1987. The Macintosh script system. *Newsletter for Asian and Middle Eastern Languages on Computer* 2(1&2):9-24.
- Ford, Ric and Connie Guglielmo. 1992. Apple's new technology and publishing strategies. *MacWeek*, (September 28, 1992), 38-40.
- Grimes, Barbara F. 1992. *Ethnologue: languages of the world (12th edition)*. Dallas, TX: Summer Institute of Linguistics.
- ISO. 1986. *Information processing--text and office systems-- Standard Generalized Markup Language (SGML)*. ISO 8879:1986 (E). Geneva: International Organization for Standardization, and New York: American National Standards Institute.
- _____. 1991. *Code for the representation of names of languages*. ISO CD 639/2:1991. Geneva: International Organization for Standardization.
- Knuth, Donald E. and Pierre MacKay. 1987. Mixing right-to-left texts with left-to-right texts. *TUGboat* 8(1):14-25.
- Rettig, Marc, Gary Simons, and John Thomson. 1993. Extended objects. *Communications of the ACM* 36(8):19-24.
- Simons, Gary F. 1989. The computational complexity of writing systems. In: Ruth M. Brend and David G. Lockwood (eds.), *The Fifteenth LACUS Forum*. Lake Bluff, IL: Linguistic Association of Canada and the United States, pages 538-553.

_____. 1994. Conceptual modeling versus visual modeling: a technological key to building consensus. A paper presented at the Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computers and the Humanities, 19-23 April 1994, Paris. Available at following Internet URL:
<http://www.sil.org/cellar/ach94.htm>.

_____. In press. The nature of linguistic data and the requirements of a computing environment for linguistic research. To appear in John Lawler and Helen Dry (eds.), *Computing and the Ordinary Working Linguist*. Mouton de Gruyter.

Sperberg-McQueen, C. M. and Lou Burnard. 1994. *Guidelines for the encoding and interchange of machine-readable texts*. Text Encoding Initiative, document number TEI P3. Sponsored by Association for Computers and the Humanities, Association for Computational Linguistics, and Association for Literary and Linguistic Computing.