Spaniel – Span-based Information Extraction Language

Adam Lally [apl2107@columbia.edu]
COMS W4115 Programming Languages and Translators
September 28, 2004

Introduction

Spaniel is a new programming language designed to support programming tasks related to information extraction. In general terms, Information Extraction is the task of building structured databases from unstructured, natural-language text. One example would be identifying named entities such as persons, places, and organizations and determining relations between them, such as which persons are employed by which organizations.

Spaniel is meant to allow average programmers to write simple information extraction programs as well as be a useful tool for the experts who practice in that field.

Background

It is common for an Information Extraction application to begin by *annotating* its raw input documents. That is, one or more components called *annotators* scan through the input document and identify *spans* of text which are labeled and assigned attributes. A labeled span is reffered to as an *annotation*. A simple annotator might take the input document:

```
John Smith works for IBM.
```

and annotate is as follows:

```
<Person gender="male">John Smith</Person> works for
<Organization type="corporation">IBM</Organization>.
```

Note that the use of XML syntax is just a convenient notation for representing annotations on spans of text, and there is no fundamental requirement to use XML for this task.

Annotators often build on the results of other annotators. For example, a second annotator might take the annotated text shown above and infer a `WorksFor` relation between John Smith and IBM. This could be recorded as an annotation over the entire sentence.

Hence the *annotation task* can be defined as: Given a text document and some (possibly empty) set of annotations over spans of that document, produce a new set of annotations that represent additional information inferred from that document. A software component that performs this task is called an *annotator*.

There are several approaches to tackling the annotation problem. Statistical annotators employ machine learning algorithms trained on human-annotated text, while rule-based annotators allow their users to declaratively specify rules for each type of annotation, which are then executed by a rule engine against each input document. These are both very active areas of current research, and have their merits. However, a currently underused approach is the procedural approach – that is, just directly writing an annotator using a procedural programming language.

Implementing an annotator directly in code is certainly possible; however one often ends up needing to write similar code in each annotator one writes, for example deciding how annotations should be represented and efficiently accessed. These issues are not as much of an issue for statistical and rule-based annotators since a single piece of software, once written, can be reapplied in many situations by training it on new data or by supplying a new set of rules.

One way to assist in the development of procedural annotators is to build a software framework that abstracts away some of these issues. In fact, this author is working on just such a project[1]. However, the Java code one writes to implement an annotator can still be somewhat repetitive – there are many patterns that reappear in each annotator one writes. This situation can be improved by building these patterns directly into the language.

Goals

Spaniel is Domain-Specific, Integrated with Java, Intuitive, and Compact, yet Readable.

Domain Specific

Spaniel is specifically designed to support the annotation task described above. The concept of a *span*, meaning a contiguous section of text, is central to the language, and spans can be manipulated with ease. Arithmetic operators can be applied to compute unions and intersections of spans. Spans can be assigned labels and attributes, and it is easy to get an iterator over spans meeting certain criteria.

While *Spaniel* does provide a basic core of programming language capability, it is not intended to be a general purpose programming language that supplants Java or C++. Developers are expected to code annotation algorithms in *Spaniel*, and use Java for other aspects of their program.

Integrated with Java

Spaniel is an interpreted language that runs within a Java Virtual Machine. As such, it is very easy for a Java program to execute an annotation algorithm written in *Spaniel* as part of a larger Java application.

What's more, the *Spaniel* language includes a way for a *Spaniel* program to make a call to a Java method. This makes the power of Java and its extensive class libraries accessible to *Spaniel* program, enabling the core annotation algorithm to be written in *Spaniel* while any complex computations are done in Java, where they belong.

Intuitive

It was decided that *Spaniel* should be a procedural language, because that is what average programmers know and can do well. While there are undoubtedly advantages to declarative and functional languages, in this author's experience average programmers are not comfortable or effective thinking in this way.

---

[1] D. Ferrucci and A. Lally. "Building an example application with the Unstructured Information Management Architecture." *IBM Systems Journal*, August 2004. http://www.research.ibm.com/journal/sj/433/ferrucci.pdf.

Anyone familiar with Java can easily learn to write programs in *Spaniel*. The Spaniel syntax is very similar to Java and the new syntax is related only to the central concepts of spans and iterators over spans, which are easy to learn.

Compact, Yet Readable

Part of the reason for creating the *Spaniel* language was to reduce some of the boilerplate code that is necessary when implementing annotation algorithms in Java. The amount of code needed to obtain iterators over spans matching certain criteria is greatly reduced. Indeed, since spans are a built-in type in the language, the amount of code for many span-based operations is reduced. This leads to more compact programs, and also contributes to readability, since a reader does not have to sift through the repetitive boilerplate code to find the important parts.

If compactness is made an end in itself, however, this gain in readability is soon dramatically reversed. *Spaniel* attempts to achieve compactness only where it increases readability. In particular, it was decided *not* to introduce a large number of new operators into the language, even for very common operations. Among the most common operations to perform on a span is to get its begin or end position as a character offset into the document; the *Spaniel* syntax for this is `span.begin` and `span.end`, not some operator form like `&span` and `#span`, which would save characters at the expense of readability.

Language Features

Regular Expression Support

It is very common to perform regular expression matching in annotation algorithms. Therefore *Spaniel* will contain built-in functions for this. As an example, the following code is all that is necessary to build a simple Phone Number annotator in *Spaniel*:

```
forAll p : matching("\((\d\d\d\)|\d\d\d-)\d\d\d-\d\d\d\d", doc)
      annotate(p, "PhoneNumber");
```

Here, the built-in `matching` function returns an iterator over spans that match the given regular expression. The `forAll` statement iterates through each match and executes `annotate`, which assigns a label to a span.

Iterators over Spans

Spaniel makes it very easy to work with iterators over spans. An example of this was already seen above, where the built-in function `matching` returns an iterator. There other built-in functions, such as `isa`, which performs the common task of retrieving all spans that have been annotated as a particular type. Functions that return iterators can also be composed to perform more complex tasks. For example, the code:

```
forAll n1 : isa(PhoneNumber, subspans(sentence))
      //do something;
```

iterates over all annotations of type `PhoneNumber` that are `subspans` of the specified `sentence`.

Span Manipulation Made Easy

Spans are a built-in data type in *Spaniel* and are easy to work with. For example, consider that given we have already annotated phone numbers (using the regular expression match from before), we now want to find phone calls between two numbers, perhaps in sentences like:

```
… phone call from 914-555-2168 to 914-555-9876 …
```

The following code does matches sentences with that pattern, given that `n1` and `n2` are the two phone numbers and `sentence` is the enclosing sentence (perhaps assigned in enclosing `forAll` loops):

```
if (matching("from",[sentence.begin, n1.begin]) &&
    matching("to'" [n1.end, n2.begin])
{
      annotate([sentence.begin, n2.end], "PhoneCall");
}
```

The syntax `[sentence.begin, n1.begin]` defines the span from the beginning of the sentence to the first phone number. Within this we search for the keyword "from." Similarly we search for the keyword "to" from the end of the first phone number to the beginning of the second phone number. If both these words are found, we annotate the span from the beginning of the sentence to the end of the second phone number as a `PhoneCall`.

Java Integration

Spaniel is great for span processing but is quite limited otherwise. To overcome these limitations, a *Spaniel* program is permitted to make an external call to a Java method. This is done through the built-in `javacall` function, for example:

```
forAll p : javacall(com.foo.MyJavaClass, x, y)
```

This would make a call to a method in the Java Class `com.foo.MyJavaClass`, which must implement a required interface defined by *Spaniel*, and pass it the arguments `x` and `y`. The Java method can return a special object representing an iterator over spans, and this iterator can be used in *Spaniel* in the same way as other iterators.

Conclusion

Spaniel is a domain-specific language for examining and annotating spans of text. It allows developers to focus on their algorithm without having to implement the details that would be necessary if implementing directly in Java; this makes code both more compact and more readable. *Spaniel* is tightly-integrated with Java, so that the full power of Java is available if needed. Finally, because it implements just a few new concepts on top of a base syntax similar to that of Java, *Spaniel* is easy to learn.