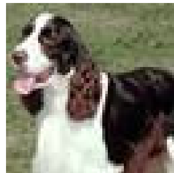
Spaniel

(Span-based Information Extraction Language)



Final Report

Adam Lally
apl2107@columbia.edu
December 21, 2004

Table of Contents

1 Introduction ... 3
1.1 Background ... 3
1.2 Goals ... 4
1.3 Language Features ... 5
1.4 Summary ... 7
2 Tutorial ... 8
2.1 Hello World in Spaniel ... 8
2.2 Tokenization and Sentence Detection ... 9
2.3 Summary ... 11
3 Language Reference Manual ... 12
3.1 Organization of this Manual ... 12
3.2 Lexical Structure ... 12
3.3 Types ... 14
3.4 The Structure of a Program ... 16
3.5 Variables ... 16
3.6 Statements ... 17
3.7 Expressions ... 18
3.8 Input and Output of a Spaniel Program ... 24
3.9 Built-in Procedures ... 25
4 Project Plan ... 28
4.1 Process ... 28
4.2 Programming Style Guide ... 28
4.3 Timeline ... 29
4.4 Software Development Environment ... 29
4.5 Project Log ... 30
4.6 Final Status ... 30
5 Architectural Design ... 31
5.1 Components of the Spaniel Interpreter ... 31
5.2 Execution ... 32
5.3 Procedure Invocation ... 32
6 Test Plan ... 34
6.1 Testing the Front End ... 34
6.2 Testing the Back End ... 34
6.3 Functional Testing ... 35
7 Lessons Learned ... 38
8 Code Listing ... 39
8.1 Overview ... 39
8.2 ANTLR Grammar Files ... 40
8.3 Package edu.columbia.apl2107.spaniel ... 47
8.4 Package edu.columbia.apl2107.spaniel.exception ... 74
8.5 Package edu.columbia.apl2107.spaniel.parser ... 76
8.6 Package edu.columbia.apl2107.spaniel.procedures ... 76
8.7 Package edu.columbia.apl2107.spaniel.test ... 85
8.8 Package edu.columbia.apl2107.spaniel.util ... 90
8.9 Package edu.columbia.apl2107.spaniel.var_value ... 95
8.10 Spaniel Test Programs for JUnit Test Case ... 106

1 Introduction

Spaniel (Span-based Information Extraction Language) 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.

1.1 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 referred to as an *annotation*. A simple annotator might take the input document:

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

and annotate it 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.

1.2 Goals

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

1.2.1 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.

1.2.2 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.

---

[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.

1.2.3 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.

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.

1.2.4 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.

1.3 Language Features

1.3.1 Regular Expression Support

It is very common to perform regular expression matching in annotation algorithms. Therefore *Spaniel* contains 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.

1.3.2 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 : instancesOf(PhoneNumber, subspans(sentence)))
      //do something;
```

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

1.3.3 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`.

1.3.4 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.

1.4 Summary

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.

2 Tutorial

This chapter will introduce you to writing programs in Spaniel.

2.1 Hello World in Spaniel

In keeping with the great computer science tradition, our first Spaniel program will be "Hello World". But instead of just printing "Hello World" to the console, we'll show something more natural to do in Spaniel – recognize occurrences of "Hello World" in a document. Here's the entire program:

```
//Hello World in Spaniel
proc main(doc)
{
  forAll (x : matching("Hello World", doc))
    annotate(x,"Greeting");
}
```

First, note that the convention for comments is the same as that in C++ and Java. The first line of this program is a comment.

All Spaniel programs must have a procedure (`proc`) named `main`. The argument to main (`doc`) represents the document to be analyzed. The `forAll` statement in this program loops over all matches of the simple regular expression `Hello World`.

Each time through the loop the variable `x` is assigned the next match. The value of the variable `x` is a value of type *Span*, which is a data structure that has a begin and end position that point back into the document text

The body of the `forAll` loop calls the built-in procedure `annotate` on each span to annotate it with the label `Greeting`. Spans are a key data type in Spaniel, and annotating spans with labels is what Spaniel programs are all about.

To run this program, first enter it into a text editor and save it to a file named `helloWorld.spl`. You'll also need a document to feed as input to the program. Create a file named `testing.txt` that contains the following:

```
This is a test... Hello World...  One more time... Hello World!
```

Now invoke the Spaniel interpreter on your program by calling the following from a command prompt:

```
>java edu.columbia.apl2107.spaniel.Interpreter –x helloWorld.spl
testing.txt
```

The output of the program will be printed to the console as an XML document (the output format was indicated by the –x option):

```
<?xml version="1.0" encoding="UTF-8"?>
<Document>This is a test... <Greeting>Hello World</Greeting>...  One
more time... <Greeting>Hello World</Greeting>!</Document>
```

Our program has successfully detected the two occurrences of the string "Hello World" in this document. The output shows this by surrounding each occurrence with a `<Greeting>` tag.

2.2 Tokenization and Sentence Detection

Very often the first step taken by a text analysis program is to divide a document into words (called tokens) and sentences. In this section will look at a Spaniel program that does this. This example will consist of multiple procedures. First, let's look just at the procedure that does tokenization:

```
//Tokenize a span of text
proc tokens(span)
{
  forAll (t : matching("([A-Za-z]+)|([\.\?!,;:\'])",span))
  {
    token = annotate(t,"Token");
    emit(token);
  }
}
```

This is much like the main procedure of the Hello World example. It uses a `forAll` statement and the built-in procedure `matching` to loop over all the Spans that match a regular expression, in this case words and punctuation. Each Span is then annotated (labeled) as a `Token`. The resulting labeled Span is assigned to the local variable `token`. Note that variables, including loop variables, do not need to be declared before they are used. They are no type declarations for arguments or variables in Spaniel, and no type checking is performed at compile time.

The main new language feature in this example is the `emit` statement. This is the method by which Spaniel procedures return values to their caller, but it is different from the `return` statement in C or Java. A call to `emit` does not cause the procedure's execution to terminate. Instead, the procedure continues executing until it completes normally (in this case, when the `forAll` loop completes). The return value of the procedure is the sequence consisting of all values that were emitted via the `emit` statement. So in this example, the return value of the `tokens` procedure is a *sequence of spans*, where each span represents a token (word or punctuation symbol) in the document.

Next, let's look at the procedure that does sentence detection:

```
//Annotate sentences
//The argument tokenStream must be a sequence of tokens
proc sentences(tokenStream)
{
  start = null;
  forAll (x: tokenStream)
  {
    if (start == null)
      start = x;
    if (matching("[.?!]",x))
    {
      emit (annotate ([start.begin, x.end],"Sentence"));
      start = null;
    }
  }
}
```

As the comment indicates, this procedure is expected to be passed a sequence of tokens as its argument. Since there are no type declarations in Spaniel, it's important to write such things in comments.

By now you should be getting the hang of using the `forAll` statement to iterate over a sequence of values. This procedure is no different – it simply iterates over all of the tokens looking for tokens that match the regular expression `[.?!]` (punctuation marks that terminate sentences). When it finds one, it creates a "Sentence" annotation.

The only piece of new syntax here is `[start.begin, x.end]`. The square brackets are used to define a new span. In this case we are defining the span that extends from the beginning of the `start` token to the end of the current token(`x`). This is the span that will be labeled as a Sentence.

Finally, we need a `main` method.

```
proc main(doc)
{
  sentences(tokens(doc));
}
```

The main method calls the `tokens` procedure, passing it the entire document is an argument. As we've seen, the `tokens` procedure returns a sequence of spans that denote the tokens of the document. This sequence of spans is then passed to the `sentences` procedure, which will iterate over them and annotate sentence boundaries.

The result of running this program on the simple document:
`This is a test. This is only a test.`

is the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Document><Sentence><Token>This</Token> <Token>is</Token>
<Token>a</Token> <Token>test</Token><Token>.</Token></Sentence>
<Sentence><Token>This</Token> <Token>is</Token> <Token>only</Token>
<Token>a</Token><Token>test</Token><Token>.</Token></Sentence>
</Document>
```

2.3 Summary

This short tutorial has covered all of the essential features of the Spaniel programming language. The key idea is that Spaniel is a language for manipulating spans (regions of a text document). We've seen how the following features are built-in to the language in order to support span manipulation:

- Regular Expression Matching (the built-in `matching` procedure)
- Iterating over Sequences of Spans (the `forAll` statement)
- User-defined Procedures that return Sequences of Spans (the `emit` statement)
- Special syntax for creating spans (square brackets)

Although not shown in the tutorial, Spaniel also overloads some operators (+, *, <, >, ==) for Spans as well. There are also several more built-in procedures that are useful for span manipulation.

Finally, should you find yourself needing to do something beyond the capabilities of Spaniel (or which is simply cumbersome to do in Spaniel), you can use the `javacall` procedure to make a call from your Spaniel program to Java code. This makes the power of Java and its extensive class libraries accessible to *Spaniel* programs, enabling the core annotation algorithm to be written in *Spaniel* while any complex computations are done in Java, where they belong.

See the Language Reference Manual for details on these additional features.

3 Language Reference Manual

3.1 *Organization of this Manual*

This manual is organized as follows:

Section 3.2 describes the lexical structure of the language, which is based on C and Java.

Section 3.3 describes the data types used in Spaniel. These consist of primitive types as well as two special types – *span* and *sequence*. Spaniel's treatment of spans and sequences of spans are what primarily differentiate it from other languages. Note that Spaniel is not a strongly-typed language, so variables do not have declared types and may take on any type of value.

Section 3.4 describes the high-level structure of a Spaniel program.

Section 3.5 describes variables in Spaniel.

Section 3.6 describes statements, which are again modeled after C and Java.

Section 3.7 describes expressions, and defines the precedence and associativity of the language's operators.

Section 3.8 describes the input and output of a Spaniel program.

Section 3.9 describes the built-in procedures that a Spaniel runtime environment is required to provide.

3.2 *Lexical Structure*

This section specifies the lexical structure of Spaniel.

3.2.1 Character Set

The character set for the Spaniel language is 8-bit ASCII, excluding the characters with ASCII codes 0, 1, and 2.

3.2.2 Line Terminators

Lines are terminated by the carriage return character ('\r'), the newline character ('\n'), or a carriage return followed by a newline, which is considered just a single line terminator.

3.2.3 Whitespace

Whitespace includes line terminators as well as the space and tab characters. Except for within String Literals, and for the fact that it separates tokens, whitespace is ignored.

3.2.4 Comments

As in Java, there are two types of comments. End of line comments begin with the characters `//` and end at the next line terminator. Block comments begin with the characters `/*` and end with the characters `*/`. Comments are ignored, except that they separate tokens.

Comments do not nest. Also, comments do not occur within String Literals.

3.2.5 Literals

There are five types of literals in Spaniel: integer literals, floating-point literals, string literals, Boolean literals, and the null literal. Each is described in a subsection below.

3.2.5.1 Integer Literals

An integer literal is a sequence of digits. Extraneous leading zeros are not permitted. The value of an integer literal is defined as the value that would be returned by the Java method call `Integer.parseInt` when passed the text of the integer literal. This is a 32-bit integer value. It is a compile time error if an integer literal causes `Integer.parseInt` to throw an exception, which would be the case for integer literals that are greater than 2147483647.

3.2.5.2 Floating-point Literals

A floating-point literal consists of an integer part, followed by a fraction part, followed by an exponent. The fraction part is a decimal point (.) followed by one or more digits. The exponent is the letter `e` followed by an optional + or – sign, followed by one or more digits. Either the integer part or the fraction part may be omitted, but not both. The exponent part may be omitted if and only if the fraction part is not omitted.

The value of a floating-point literal is defined as the value that would be returned by the Java method `Float.parseFloat` when passed the text of the floating-point literal. This is a 32-bit IEEE floating-point value. It is a compile time error if a floating-point literal causes `Float.parseFloat` to throw an exception.

3.2.5.3 String Literals

There are two types of String Literals:

(1) A sequence of characters enclosed in double-quotes. The value of such a string literal is the string consisting of the exact characters so-enclosed, with the exception that to represent a double quote character within a string literal, two consecutive double-quote characters are used. Line separator characters may not be included in this type of string literal.

(2) A sequence of character codes, where each character code consists of the `#` character, followed by one or two hexadecimal digits. The value of each character code is the character whose ASCII value is represented by the hexadecimal digits, and the value of the string literal is the string of such values.

This form is necessary to represent linefeed and carriage return characters (#A and #D, respectively) within string literals.

3.2.5.4 Boolean Literals

The character sequences `true` and `false` represent literal Boolean values

3.2.5.5 Null Literal

The character sequence `null` represents the null value.

3.2.6 Keywords

The following character sequences are keywords:

```
break
else
emit
forAll
if
proc
while
```

3.2.7 Identifiers

Identifiers are character sequences beginning with a letter or underscore and consisting of letters, underscores, and digits, not including keywords or the literals `true`, `false`, or `null`.

3.2.8 Separators

The following characters are used as separators in the syntax definitions of statements and expressions:

```
(    )    [    ]
{    }    ,    ;
.    :
```

3.2.9 Operators

The following characters or character sequences are operators. Their meanings, associativity, and precedence are defined in Section 3.7.

```
<    <=   >    >=
+    -    *    /
%    &&   ||   !
=    ==   !=
```

3.3 Types

Spaniel is not a strongly-typed language, so there are no type definitions in the syntax of the language. However, to understand the semantics of the language it is important to understand the types to which expressions can evaluate. The following types exist in Spaniel:

- Integer: 32-bit signed integer value
- Float: 32-bit IEEE floating point value
- String: sequence of ASCII characters
- Boolean: true or false
- Span (see section 3.3.1)
- Sequence (see section 3.3.2)

In addition, expressions can evaluate to null, which indicates the lack of a value. The null value is not of any type.

3.3.1 Span

Conceptually, a span represents a region of a text document to which information is attached. Spaniel defines this as a core type because it is useful in the information extraction applications that Spaniel is designed to support.

Specifically, a *span* (sometimes called a *span object*) is a structure with arbitrarily many fields, but with three specific field names reserved: `begin`, `end`, and `type`.

The fields `begin` and `end` are expected to hold integer values that hold the start and end character offsets of the span. The field `type` is expected to hold a String that identifies what the span represents. Attempting to assign the wrong type of value to one of these fields results in a run-time error, an exception to the general lack of type checking in Spaniel.

3.3.2 Sequence

A sequence is an ordered collection of values. The values in a sequence need not all be of the same type, and a sequence may contain other sequences, to arbitrary levels of nesting.

Spaniel has some unusual properties involving sequences. For one, all Spaniel procedures return sequences (see section 3.6.7). Also, in Spaniel expression evaluation, sequences of a single element, in most contexts, are treated equivalently to the single element alone (see section 3.3.3.1).

3.3.3 Type Coercion

In Spaniel, there are two cases where values of one type must be *coerced* to a different type.

3.3.3.1 Single-Element Sequence Coercion

Sequences of a single element, in most contexts, are treated equivalently to the single element alone. Anywhere that a non-sequence value is required, any sequence value consisting of exactly one element must be a coerced to that one element.

This value coercion is important in light of the fact that all Spaniel procedures technically return sequences.

3.3.3.2 Boolean Coercion

Anywhere that a Boolean-typed value is required, the following values must be coerced to a Boolean-typed value as indicated:

- A sequence containing exactly one element, which is of type Boolean, is equivalent to its element. (This is a restatement of the single-element sequence coercion rule.)
- An empty sequence is equivalent to the Boolean value `false`.
- Any other sequence is equivalent to the Boolean value `true`.

3.4 The Structure of a Program

A Spaniel *program* is one or more procedures. Each procedure begins with the keyword `proc`, followed by the procedure's name (an identifier), a list of formal parameters (zero or more comma-separated identifiers), and the body of the procedure (a block, which is a series of statements enclosed in curly braces). For example:

```
proc foo(a,b)
{
  //statements go here
}
```

Each procedure in a valid Spaniel program must have a unique name, and that name must not be the same as any of the built-in procedures defined in section 3.9. A valid Spaniel program must contain a procedure whose name is `main`, which is the initial procedure invoked by the Spaniel interpreter when the program is run. The main procedure must take at least one parameter. It is a compile time error if a Spaniel program does not meet these constraints.

3.5 Variables

A *variable* is a storage location that can take a value of any of the types listed in section 3.3. In Spaniel, variables do not have declared types; any variable may take on any value. Generally, a variable is created when it is referenced for the first time.

There are three types of variables:

- *Procedure Parameter Variables:* each time a procedure is invoked, a new variable is created for each formal parameter in that procedure declaration. These variables have the same name as the formal parameters, and their initial values are taken from the actual parameters in the procedure invocation.
- *Local Variables*: each time a procedure is invoked, a new variable is created for each variable (the leftmost identifier in each lvalue expression) referenced in the procedure's body. The initial value of a local variable is always the null value.
- *Field Variables:* a field is a named slot within a value of type Span, as defined in section 3.3. A field variable is created whenever a field is referenced within an lvalue expression (section 3.7.1.1), and no field with that name already exists within the span. The initial value of a field variable is always the null value.

Procedure parameter variables and local variables exist only within the invocation of the procedure in which they were created. They cease to exist when execution of the procedure terminates.

Field variables exist as long as they are referenceable (which is as long as the span object containing the field variable is referenceable). A Spaniel interpret should arrange to garbage collect unreferenceable span objects and their field variables.

3.6 Statements

3.6.1 Block

A block is zero or more statements enclosed in curly braces. A block is executed by executing each of its statements, in order, unless otherwise indicated.

3.6.2 Expression Statement

Two types of expressions – assignments (section 3.7.8) and procedure calls (section 3.7.1.4) – can be used as statements. The expression must be followed by a semicolon.

When the statement executes, the expression is evaluated, causing any side effects it may have to take place.

3.6.3 If Statement

There are two forms of the if statement:

`if (`*expression*`)` *statement*

`if (`*expression*`)` *statement* `else` *statement*

An `else` clause is always associated with the most recently encountered elseless if.

When an if statement executes, the expression is evaluated. If it evaluates to `true`, the first statement is executed. If it evaluates to `false`, the statement after the else, if present, is executed. If the expression evaluates to a non-Boolean value, it is a runtime error.

3.6.4 ForAll Statement

The forAll statement has the form:

`forAll (`*identifier* **:** *expression*`)` *statement*

The expression is evaluated. If it does not evaluate to a sequence, it is a run-time error. Otherwise, let `s` equal the sequence to which it evaluates, and execute the following:

(1) If s is an empty sequence, the forAll statement's execution completes.
(2) Assign the first element of `s` to the variable named by the identifier in the forAll statement
(3) Execute the sub-statement
(4) Let the new value of s be the sequence formed by removing the first element from the current value of s.

(5) Go to step 1.

3.6.5 While Statement

The while statement has the form:
`while (`*expression*`)` *statement*

Execution of a while statement is as follows:

(1) Evaluate the expression. If it evaluates to false, the while statement's execution completes.
(2) Execute the sub-statement.
(3) Go to step 1.

3.6.6 Break Statement

The keyword `break`, followed by a semicolon, is a statement whose execution causes the immediate termination of the immediately enclosing forAll or while statement.

A break statement outside of a forAll or while statement causes a compile-time error.

3.6.7 Emit Statement

The emit statement has the form:
`emit` *expression* `;`

When an emit statement executes, its expression is evaluated. The resulting value is appended to the sequence of values that the currently executing procedure will return when it terminates. A consequence of this is that *all* Spaniel procedures return values of type sequence.

3.6.8 Empty Statement

A semicolon by itself is a valid statement, whose execution does nothing.

3.7 Expressions

Expressions are parts of a program that can be *evaluated* to produce a value. When expressions are evaluated, they may also cause *side effects* to take place. Ultimately it is the side effects of expressions that perform the work of the program and generate its output. Expressions always occur within statements; the specification for statements in section 3.5 defines when expressions are to be evaluated.

Most expressions, when evaluated, produce (evaluate to) a value of one of the types defined in section 3.3. The only exception is *lvalue expressions* (section 3.7.1.1), which evaluate to variables.

The precedence of operators is the same as the major subsections of this section (highest precedence first). Thus the operands to + (Section 3.7.4) are those expression types defined in Sections 3.7.1 through 3.7.3.

3.7.1 Primary Expressions

Primary expressions are the core expressions from which other expressions are built. They include lvalue expressions, literals, span expressions, procedure calls, and parenthesized expressions.

3.7.1.1 Lvalue Expressions

An *lvalue expression* is either an identifier alone, or an lvalue expression followed by a dot followed by an identifier. Examples include:

```
foo
foo.bar
foo.bar.baz
```

Lvalue expressions are so named because they can appear on the left side of an assignment. That is, lvalue expressions evaluate to *variables*, which are storage locations to which values can be assigned.

An lvalue expression consisting of an identifier alone evaluates to the variable whose name is that identifier (see section 3.5).

An lvalue expression of the form *lvalue_exp* **.** *identifier* is evaluated by first evaluating the l*value_exp* to the left of the dot. By definition, this will evaluate to a variable. Let v be the value of this variable. If v is not of type span, it is a runtime error. Otherwise, the lvalue expression evaluates to the variable that is the field of span v that is named by the *identifier*.

For simplicity, Spaniel uses a more restricted syntax for lvalues than other languages such as C and Java. Lvalues in Spaniel can only contain identifiers separated by dots, thus for example `foo(x).field` is not valid in Spaniel. This turns out not to be a very useful expression anyway, because of the fact that procedures in Spaniel always return sequences, which are not lvalues.

In the remainder of this specification, unless otherwise indicated, the terminology "the value of an expression" or "the value to which an expression evaluates," when applied to lvalue expressions, should be taken to mean the value of the variable to which the lvalue expression evaluates.

3.7.1.2 Literals

The syntax for literals is defined in section 3.2.5.

Integer, Floating-point, and String Literals evaluate to the values defined in section 3.2.5, which are of type Integer, Float, and String, respectively.

The literals `true` and `false` evaluate to the Boolean-typed values true and false, respectively.

The literal `null` evaluates to the null value.

3.7.1.3 Span Expressions

A span expression is of the form:
`[`*expression*`,` *expression*`]`

When a span expression is evaluated, a new value of type span (the "new span object") is created. The value of the first sub-expression is then assigned to the `begin` field variable of the new span object, and the value of the second sub-expression is assigned to the `end` field variable of the new span object. The `type` field variable of the new span object is set to the null value.

3.7.1.4 Procedure Calls

A procedure call expression is of the form:
identifier `(` *expression_list* `)`

It is a compile-time error if the identifier that begins a procedure call expression does not match the name of any defined procedure, either built-in (see section 3.9) or declared within the user's program (see section 3.4), or if the number of arguments in the procedure call expression does not match the number of arguments in the procedure declaration. Otherwise, the procedure so named is referred to as the "named procedure" in the description below:

When a procedure call expression is evaluated, first each of the sub-expressions is evaluated, in order. Then the named procedure is invoked, with actual parameters equal to the values to which the sub-expressions evaluated. The procedure call expression evaluates to the value returned by this invocation, which is always of type Sequence.

3.7.1.5 Parenthesized Expressions

An expression enclosed in parentheses evaluates to the same variable or value that the enclosed expression evaluates to.

3.7.2 Unary Expressions

There are three unary operators in Spaniel: +, -, and !. A unary expression is defined to be one of these operators, followed by another unary expression. The unary operators associate to the right.

Unary expressions are evaluated as follows:

+ expression
The operand must evaluate to type Integer or Float, otherwise it is a run-time error. The unary expression evaluates to the same value as its operand.

– *expression*
The operand must evaluate to type Integer or Float, otherwise it is a run-time error. The unary expression evaluates to the arithmetic negation of its operand. If overflow occurs, it is a run-time error.

! *expression*
The operand must evaluate to type Boolean, otherwise it is a run-time error. For a Boolean-typed operand, the unary expression evaluates to the Boolean negation of its operand. (i.e. !true evaluates to false and !false evaluates to true).

3.7.3 Multiplicative Expressions

The multiplicative operators are *, /, and %. They associate to the left. Multiplicative expressions are evaluated as follows:

expression* * *expression
If both operands evaluate to integers, then the expression evaluates to the result of performing integer multiplication on the operand values.

If one operand evaluates to a float and the other to an integer or a float, then the expression evaluates to the result of performing floating-point multiplication on the operand values.

If both operands evaluate to spans, then the expression evaluates to the *intersection* of the spans. That is, [a,b]*[c,d] evaluates as follows:

- If c>b or a>d, the expression evaluates to null
- Otherwise, the expression evaluates to [max(a,c), min(b,d)].

In all other cases, the result is a run-time error.

expression* / *expression
If both operands evaluate to integers, then the expression evaluates to the result of performing integer division on the operand values.

If one operand evaluates to a float and the other to an integer or a float, then the expression evaluates to the result of performing floating-point division on the operand values.

In all other cases, the result is a run-time error.

expression* % *expression
If both operands evaluate to integers, then the expression evaluates to the result of performing the modulo operation on the operand values.

In all other cases, the result is a run-time error.

3.7.4 Additive Expressions

The additive operators are + and -. They associate to the left. Additive expressions are evaluated as follows:

expression + expression
If both operands evaluate to integers, then the expression evaluates to the result of performing integer addition on the operand values.

If one operand evaluates to a float and the other to an integer or a float, then the expression evaluates to the result of performing floating-point addition on the operand values.

If both operands evaluate to spans, then the expression evaluates to the *minimal enclosing span* of the operands. That is, [a,b]+[c,d] evaluates to [min(a,c), max(b,d)].

If both operands evaluate to Strings, then the expression evaluates to the concatenation of the two Strings. If the left operand evaluates to a String and the right operand evaluates to an Integer or Float, then for the + operator, the expression evaluates to the concatenation of the String with the String representation of the Integer or Float.

In all other cases, the result is a run-time error.

expression − expression
If both operands evaluate to integers, then the expression evaluates to the result of performing integer subtraction on the operand values.

If one operand evaluates to a float and the other to an integer or a float, then the expression evaluates to the result of performing floating-point subtraction on the operand values.

3.7.5 Relational Expressions

The relational operators are ==, !=, <, <=. >, and >=. They do not associate, and always evaluate to a value of type Boolean. Relational expressions are evaluated as follows:

expression == expression
The expression evaluates to true if both of its operands evaluate to the same type and have values that are equal. Otherwise it evaluates to false. For spans, equal means having begin and end values that are equal. For sequences, equal means that all elements of the first sequence equal all elements of the second sequence, in order.

expression != expression
The expression evaluates to true if == would have evaluated to false, and evaluates to false if == would have evaluated to true.

For the remaining relational operators, it is a run-time error if either operand evaluates to type Boolean, String, or Sequence, or if one operand evaluates to type Span and the other does not.

expression < expression
The expression evaluates to true if its first operand is less than its second operand. Otherwise it evaluates to false. For spans, s1 < s2 if and only if (s1.begin < s2.begin) or (s1.begin == s2.begin and s1.end > s2.end). The greater-than symbol is correct; the consequence of this is that for spans with the same begin value, longer spans are less than shorter spans, and thus appear first in an ordered sequence of spans, which is a desirable property for iteration over nested spans.

expression > expression
The expression evaluates to true if its first operand is greater than its second operand. Otherwise it evaluates to false. For spans, s1 > s2 if and only if (s1.begin > s2.begin) or (s1.begin == s2.begin and s1.end < s2.end). The consequence of this is that for spans with the same begin value, shorter spans are greater than longer spans, which is consistent with the definition of the < operator.

expression <= expression
The expression evaluates to true if either < or == would have evaluated to true, and false otherwise.

expression >= expression
The expression evaluates to true if either > or == would have evaluated to true, and false otherwise.

3.7.6 And Expressions

The And operator is `&&`. It is left associative. An And expression has the form
expression && expression

and is evaluated as follows:

The first operand is immediately evaluated. If it does not evaluate to a Boolean value, it is a run-time error.

If the first operand evaluates to false, the And expression also evaluates to false. The second operand expression is not evaluated in this case.

If the first operand evaluates to true, the second operand expression is then evaluated. If this evaluates to a non-Boolean value, it is a run-time error. Otherwise, the And expression evaluates to the same value as the second operation expression.

3.7.7 Or Expressions

The Or operator is `||`. It is left associative. An Or expression has the form
expression || expression

and is evaluated as follows:

The first operand is immediately evaluated. If it does not evaluate to a Boolean value, it is a run-time error.

If the first operand evaluates to true, the Or expression also evaluates to true. The second operand expression is not evaluated in this case.

If the first operand evaluates to false, the second operand expression is then evaluated. If this evaluates to a non-Boolean value, it is a run-time error. Otherwise, the Or expression evaluates to the same value as the second operation expression.

3.7.8 Assignment Expressions

The Assignment operator is =. It is right associative. An assignment expression has the form
lvalue_expression = expression

When an assignment expression is evaluated, its right operand is first evaluated. The value of this expression is then assigned to the variable to which the left operand (which must be an lvalue expression) evaluates. The assignment expression then evaluates to that same value.

3.8 Input and Output of a Spaniel Program

The Spaniel language is designed to support the *annotation task*, which is defined as follows: 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.

As such, the input to a Spaniel program always includes an *annotated document*, which is logically just a character string, along with zero or more span objects (see section 3.3). The actual representation of the input is not relevant to the Spaniel programmer, and this choice is left up to the Spaniel interpreter. One choice is XML; that is, the following string might be the input to the interpreted Spaniel program:

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

This represents the string "John Smith works for IBM" and two span objects, one with fields begin=0, end=10, type="Person", and gender="male", and the second with fields begin=21, end=24, type="Organization," and type="corporation."

When the `main` procedure of the Spaniel program is invoked, it is always passed, as its first argument, the span that defines the entire implicit annotated document; that is, `begin` = 0, `end` = the length of the document text, and `type` = "Document". The

contents of the annotated document can then be accessed by passing this span object to the built-in functions described in the next section.

The output of a Spaniel program also always includes an annotated document. For example, a Spaniel program that took the above input might infer a "Works For" relation between John Smith and IBM, and represent that as an annotation over the entire span of the document. This could then be written out to an XML representation similar to that shown above. Spaniel programs post annotations to the output document using the `annotate` built-in procedure defined in section 3.9.

The input to a Spaniel program may include other arguments, in addition to the annotated document, that can be used to parameterize the behavior of the program. Also, the output of a Spaniel program can include console output generated by calls to the `print` and `println` procedures defined in section 3.9, as well as arbitrary output generated by calls into Java, which are made using the `javacall` procedure also defined in section 3.9.

3.9 Built-in Procedures

The following procedures are required to be built-in to any Spaniel interpreter. It is a compile-time error if a user's program declares a procedure with any of these names. As previously noted, all Spaniel procedures return sequences.

3.9.1 first

If `s` is a value of type Sequence that has at least one element, `first(s)` will return the sequence containing only the first element of `s`. Otherwise, it will return an empty sequence

3.9.2 rest

If `s` is a value of type Sequence, `rest(s)` will return the sequence containing all elements of `s` except the first. If `s` is an empty sequence, an empty sequence will be returned.

3.9.3 print

`print(v)` will print a string representation of `v` to the console (technically, to whatever output stream the interpreter dictates). If `v` is of type string, `v` itself will be printed. Otherwise, the interpreter should convert the value to a string in whatever way it chooses. The empty sequence is returned.

3.9.4 println

`println(v)` will print a string representation of `v` to the console (technically, to whatever output stream the interpreter dictates), followed by a newline. If `v` is of type string, `v` itself will be printed. Otherwise, the interpreter should convert the value to a string in whatever way it chooses. The empty sequence is returned.

3.9.5 annotate

`annotate(s,t)` performs two operations. First, it assigns `t` to `s.type`. (That is, it labels a span.) Then, it posts s to the implicit annotated document, so that it will be in the output of the program, and will be accessible to the built-in functions that examine the implicit annotated document. The sequence containing `s` is returned.

As specified in the definition of the span type and the definition of the lvalue expression (dot operator), it is a runtime error if `s` is not a span or `t` is not a string. Additionally it is a runtime error if the begin and end values do not allow it to be posted to the implicit annotated document (i.e. if begin > end, or begin < 0, or end > length of document).

3.9.6 reMatch

`reMatch(r,s)` searches for the first match of regular expression r over the span s (that is, between character positions `s.begin` and `s.end` of the implicit annotated document). If match is found, a new span object is created, its begin field is set to the first character position of the match, and its end field is set to one plus the last character position of the match. In addition, its `_group` field is set to the number of the top-level group within the regular expression that matched. For example, if `r == "(foo)|(bar)"`, then `_group` will be set to 1 if "foo" matched and to 2 if "bar" matched. `reMatch(r,s)` returns the sequence containing the new span object as its only element. If no match is found, `reMatch(r,s)` returns the empty sequence.

It is a runtime error if r is not a string or could not be parsed as a regular expression, or if s is not a span or is an invalid span (i.e. if begin > end, or begin < 0, or end > length of document).

3.9.7 matching

`matching(r,s)` returns the sequence containing all non-overlapping matches of regular expression r over the span s, in order of increasing start position. This procedure is built-in for convenience purposes, as it can be easily implemented in Spaniel as:

```
proc matching(r,s)
{
  nextMatch = reMatch(r,s);
  while(nextMatch)
  {
    nextStart = nextMatch.end;
    emit nextMatch;
    nextMatch = reMatch(r,[nextStart,s.end]);
  }
}
```

3.9.8 subspans

If `s` is a span, `subspans(s)` returns the sequence containing all subspans of s that are contained in the implicit annotated document. Here, a subspan of s is defined as a span x such that `x.begin` >= `s.begin` and `x.end` <= `s.end`. The order of the sequence is

such that each element is always >= the previous element. (See the definition of <= for spans in section 3.7.5.)

3.9.9 instancesOf

If `s` is a span, `instancesOf(t,s)` returns all subspans `x` of `s` such that `x.type` == `t`. It is built-in for convenience purposes, as it can easily be implemented in Spaniel as:

```
proc instancesOf(t,s)
{
  forAll(x : subspans(s))
  {
    if (x.type == t)
      emit x;
  }
}
```

3.9.10 javacall

The special procedure `javacall` allows a Spaniel program to make a call to a Java method. Unlike other Spaniel procedures, `javacall` does not have a fixed number of arguments. Also, the first argument to `javacall` must be a String literal.

`javacall(c,a1,a2,…,an)` executes by first instantiating a Java class named `c`, which must implement the interface `edu.columbia.apl2107.spaniel.JavaProcedure`. Second, the `init()` method of that class is called, passing it the Java array {a1', a2', …, an'}, where a_i' is the java equivalent of the Spaniel value a_i (this is to be formally defined in the JavaDocs for the `JavaProcedure` interface).

Then, the `next()` method of that class is called to produce the next value in the sequence that `javacall(c,a1,a2,…an)` will evaluate to. When `next()` returns `null`, there is no next element. This step by step evaluation is done to support both procedure execution styles specified in section 5.3.

An interpreter implementation may, but need not, choose to implement all of the built-in functions in this section using the `javacall` mechanism.

4 Project Plan

4.1 Process

The project was divided into two development phases: front-end and back-end. The front-end was completed (save some very minor adjustments) before any back-end work commenced. The primary reasons for this clear separation were that it was desirable to have a working front end by the Language Reference Manual due date, so that I could be sure that what was written in the LRM would actually work, and that it was helpful to wait for course lectures to cover some back end topics before attempting to design a back end.

Within each phase, however, an iterative development process was employed. That is, a very simple system was first designed, implemented, and tested to completion. Then, repeated passes were made in order to refine, correct, and enhance the system. Each pass consisted of design work, implementation, and testing. This approach is valuable because it encourages stability throughout the development process – from the first iteration you have a working, tested system, and you are never in the position that 90% of the code is written but you don't have anything working to show for it

4.2 Programming Style Guide

4.2.1 ANTLR Style Guidelines

Two spaces should be used for indentation. Tabs are not used since they display differently in different editors.

An ANTLR rule with a single alternative on the right-hand side and no action should be written on a single line, as in:

```
procedure: "proc"^ ID formal_params  block;
```

For an ANTLR rule with multiple alternatives, he first line should contain the left-hand-side nonterminal symbol by itself, the next line should be a colon(:) followed by the first alternative, and each subsequent line should be a vertical bar(|) followed by the next alternative. The final line should be a semicolon(;) by itself. All lines but the first should be indented two spaces. For example:

```
primary_exp
  : lvalue_exp
  | literal
  | proc_call_exp
  | span_exp
  | LPAREN! expression RPAREN!
  ;
```

Where there is an action associated with an alternative, it should be listed immediately beneath that alternative, indented four spaces so that they line up with the beginning of that alternative, not the | symbol. For example:

```
program:
  : (procedure)+
    {#program = #([PROGRAM],program);}
  ;
```

If the action consists of more than one line of code, then the Java code conventions should be followed. The open and close braces of the action should each be on their own line in this case. For example:

```
protected CharacterCode
  : '#'! HexDigit (HexDigit)?
    {
      //set the text of this rule to the character having this
      //ASCII value
      char c = (char)Integer.parseInt($getText,16);
      $setText(c);
    }
  ;
```

4.2.2 Java Style Guidelines

Two spaces should be used for indentation. Tabs are not used since they display differently in different editors.

All Spaniel classes should be in the package `edu.columbia.apl2107.spaniel` or a subpackage thereof.

Class names should begin with capital letters, method and local variable names should begin with lowercase letters. If the name is composed of multiple words, each word after the first should begin with a capital letter.

Field names should begin with an underscore(_) to distinguish them from local variables.

Open and close braces should always appear on a line by themselves.

Meaningful JavaDoc comments should be provided for all nontrivial methods.

4.3 Timeline

Date	Target
9/28	White Paper Complete
10/21	LRM, Lexer, and Parser (no AST building) Complete.
11/12	AST Builder Complete
12/12	Back-end Feature-Complete
12/21	Demoable system and Final Report Complete

4.4 Software Development Environment

Spaniel was implemented entirely in Java 1.4, with the help of lexer and parser code generation using ANTLR v2.7.4. Some test cases were built using JUnit v3.8.1.

All development was done in the Eclipse open-source IDE (http://www.eclipse.org), v3.0, running on a Windows XP operating system. No version control system was employed, but the machine is backed up daily to guard against any catastrophic data loss.

4.5 Project Log

Date	Status
9/20	Initial Language Design
9/27	White Paper Complete and Submitted
10/07	Lexer Complete and Tested on Sample Programs
10/20	Parser Complete (no AST building), LRM v1 Complete and Submitted
11/04	ANTLR AST Built. Begin work on tree walker to convert to Custom AST classes which will constitute the back-end.
11/13	Implemented simple expressions, Integer and String types, Variable assignment, and print procedure. First instance of a running Spaniel program
11/20	Implementations of most kinds of expressions and non-recursive procedure calls
12/05	Recursive functions implemented
12/07	Sequence and Span Types implemented
12/09	All Statements and Expressions Implemented
12/10	Implemented AnnotatedDocument
12/13	Implemented Regular Expression Matching, can now run PhoneCallAnnotator example from tutorial.
12/14	Finished Implementing Built-in Procedures. Built a JUnit test case. Updated/fixed LRM based on experiences during implementation.
12/16	Implemented XML Generation in Interpreter. Completed Tutorial.
12/18	Final Report Complete

4.6 Final Status

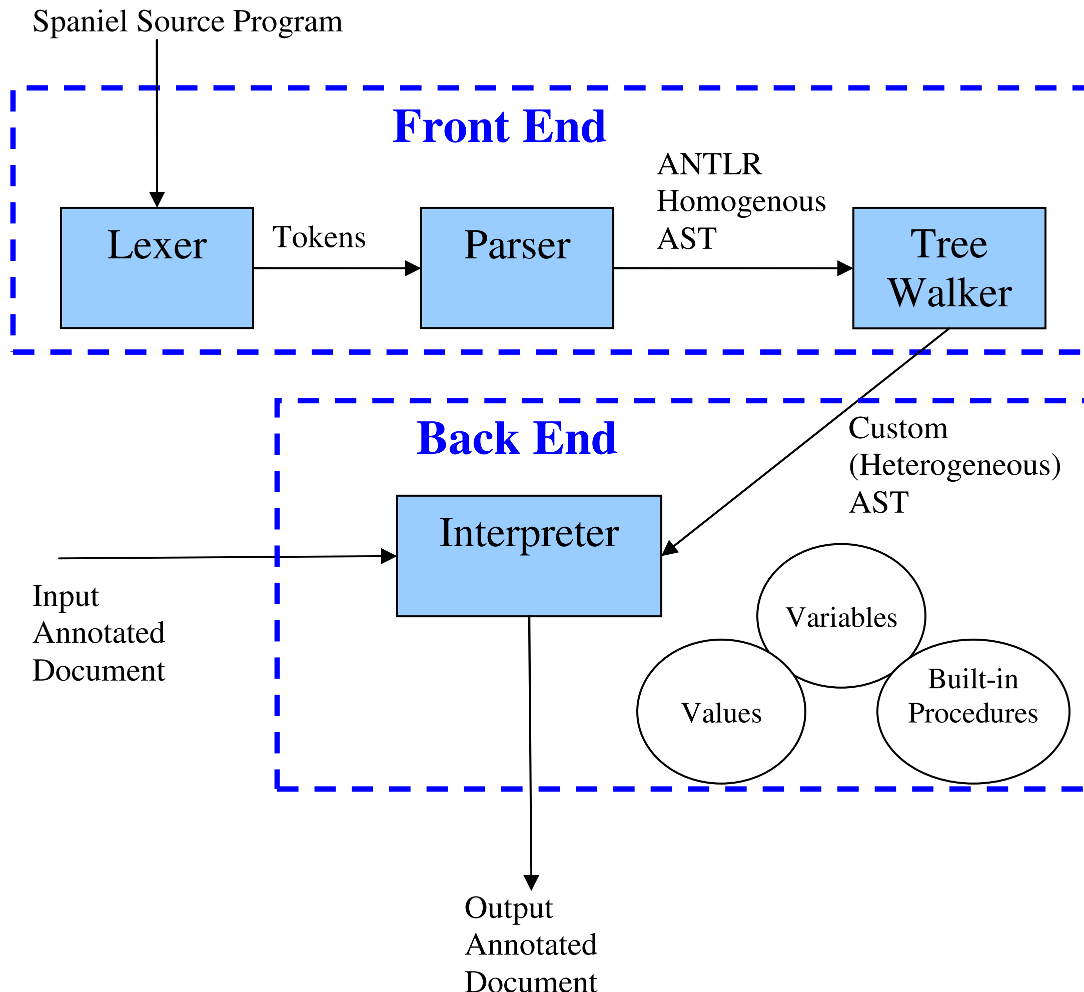
All features specified in the LRM have been implemented. A good deal of testing has been done, and there are no known bugs.

The two main shortcomings of the current implementation, due to lack of time, are error reporting and performance. Runtime error messages do not give line numbers and hence are difficult to track down. Very little attention was given to performance during the implementation of the back end, and no performance analysis was done. If there were another member on my team, I would have had him or her work on boosting the speed of the interpreter.

5 Architectural Design

5.1 Components of the Spaniel Interpreter

The following diagram illustrates the components that make up the Spaniel interpreter:



The Spaniel interpreter consists of two main modules – a front end and a back end. The front end is responsible for taking the Spaniel Source Program and producing an intermediate representation, which in this case is the "Custom AST." The Custom AST is composed of instances of Spaniel-specific Java classes. Generally speaking, there is one class for each of the various Statement and Expression types defined in the Language Reference Manual.

The back-end consists of the Interpreter proper plus the implementations of Variables, Values, and Built-in Procedures. The interpreter takes a custom AST plus an Input Annotated Document, and produces an Output Annotated Document. This is specified in more detail in the following sections.

5.2 Execution

The following steps take place when a Spaniel program is run:

1. The Spaniel program's source code is parsed, building a custom AST over which the rest of execution will operate. If any syntax errors are detected, they are reported and execution ends.
2. Static semantics checks are performed. These include checks that the following are true:
   - There is a main procedure declared, and it has at least one argument
   - No procedure is defined more than once
   - No procedure is defined using the same name as a built-in procedure
   - All procedure call expressions refer to defined procedures (either built-in or user-defined)
   - break statements do not occur outside of loops

   If any of these conditions do not hold, an error is reported an execution ends.
3. The input to the program, which currently must be a text file, is read and converted to an internal annotated document data structure, called the *implicit annotated document*.
4. The `main` procedure of the Spaniel program is invoked, passing the arguments determined by the input. The first argument is always the span that defines the entire implicit annotated document; that is, `begin` = 0, `end` = the length of the document text, and `type` = "Document". Procedure invocation and execution is described in the next section.
5. When execution of `main` completes, the implicit annotated document, possibly modified by the execution of the program, is written out using any format and output stream chosen by the interpreter, and execution of the program ends.

5.3 Procedure Invocation

Procedure invocation and execution in Spaniel are somewhat unusual. The basic idea is that there is no `return` statement that signals the completion of the procedure and specifies a single returned value. Instead, there is an `emit` statement that specifies one value in a sequence of values to be returned from the procedure. Conceptually, procedure execution continues past the `emit` statement until the procedure completes normally, by executing the last statement in the procedure's block of statements (body). At that point, the "return value" of the procedure is the sequence containing the values that were emitted by the `emit` statements, in the same order in which those `emit` statements were executed.

Specifically, procedure invocation is defined as follows:

When a procedure call expression (section 3.7.1.4) is evaluated, a new *activation record* (this is logically similar to an activation record in other languages, but there are differences in the way it may be used.) Procedure parameter variables and local variables are created within that activation record. The procedure parameter variables are initialized to the values passed from the procedure call expression, and the local variables are initialized to null.

There are then two options for an interpreter to choose to implement procedure execution:

(1) Transfer control to the invoked procedure and execute its method body. Each time an emit statement is executed, add the emitted value to an implicit sequence-typed variable representing the return value of the procedure (contained within the activation record). When the method body execution completes, return control to the procedure call expression, whose evaluation then completes; the procedure call expression evaluates to the sequence of emitted values. The activation record is then deallocated.

(2) The procedure call expression immediately completes, and evaluates to an *active sequence* object which contains a reference to the invoked procedure, its activation record, and current instruction pointer (initially pointing at the first statement in the procedure body). Then, each time the next element is requested from the active sequence object, run the invoked procedure until either its next emit statement executes or its execution completes. If an emit statement is executed, that is the next value in the active sequence. If the procedure execution completes, there is no next value in the active sequence.

The current interpreter implementation uses choice #1, for simplicity. Spaniel procedures were designed to support implementation choice #2, because it is thought to be useful in many applications of annotating a document. For example, you may want to tokenize the document and do something with each token. Rather than tokenizing in one pass and then iterating back through the tokens, you can write a procedure tokens() that emits tokens, and another procedure can execute forAll(t:tokens()), and then do something with each token t emitted by the tokens() procedure, after which each token can be discarded.

6 Test Plan

6.1 Testing the Front End

Throughout the iterative development of the front end, sample Spaniel programs were used to test the lexer and parser. These programs started out very simple and became more complex as more features were implemented in the parser. A Java driver program `TestParser` was written which would read all input programs from a directory, run them through the parser, and write the resulting ASTs to output files in XML format. This test was run often during parser development to verify that the sample programs still parsed correctly. Verification of correct ASTs was done manually at this point.

The following is an example of a (semantically nonsensical) program used to test the parsing of various lexical and syntactic constructs:

```
/* This is an example
   of a valid spaniel program. */
proc main(doc)
{
       foo = bar7; //simple assignment
       foo = 3 + 5.6 * 8e3; /* arithmetic expression involving numeric
literals*/
       bar = "\back\slashes\are\ok\\\\\" + /* string concatenation! */
             """Blah"", he said.";
       print("hello, world"); //function call
       print(#A#D); //newline
       foo = sqrt(3.4*8.7) + (6 - 2) + foo/8.4e-2*log(pi); //more complicated
       foo = [3, 4];  //span definition
       doSomething([foo=3,bar]); //another span definition
       print("if"); //string literals should not be taken as keywords
       if (foo == 3)
       {
         print("woo hoo!");
         foo(19);
       }
       else
       {
         print("oh well");
         foo(0);
       }
       if (w) if (x) y(); else z(); //dangling else
       javacall("edu.columbia.apl2107.spaniel.procedures.Println",foo(x));
       foo = "/*this is a string literal, not a comment */, //right?";
       x.y.z = a.b.c.d;
}


//omitted to save space -- proc definitions to satisfy static semantics
```

6.2 Testing the Back End

The earliest iterations of the back end were tested by writing Spaniel programs that printed output to the console, and verifying expected behavior by hand.

However, there was a great opportunity for automated testing after completing the implementation of Spaniel's `javacall` mechanism, which allows a Spaniel program to make a call to an External Procedure written in Java. I implemented External Procedures that hooked into JUnit (a unit testing utility for Java). Essentially this allowed me to write test cases directly as Spaniel programs. For example, here is a simple test program for recursive procedures:

```
/* Test program for recursive procedures, intended to be run as part of JUnit
test case. */
proc main(doc)
{
  assertEquals(1, fact(0));
  assertEquals(6, fact(3));
  assertEquals(120, fact(5));
  assertEquals(3628800, fact(10));
}

proc fact(n)
{
  if (n < 2)
    emit 1;
  else
    emit n * fact(n-1);
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

This program can be easily executed from a JUnit test case. If any of the `assertEquals` tests fails, it will cause the JUnit test case to fail and an error to be reported.

As additional features were added to the back end, test cases were written and added to the suite, and the entire suite was re-run as a regression test to catch new bugs that might have been introduced.

6.3 Functional Testing

As the Spaniel interpreter neared completion, it could be tested on more serious programs. A significant milestone was when the following program, which is very similar to the example included in the original version of the LRM, could be executed successfully. The intent of this program is to tokenize text, then detect sentence boundaries, then detect phone numbers, and finally to detect mentions of phone calls between two phone numbers, using a very simple rule.

```
proc tokens(span)
{
  tokenPattern = "([A-Za-z]+)|([\.\?!,;:\'])";
  forAll (t : matching(tokenPattern,span))
  {
    token = annotate(t,"Token");
    //the "matching" procedure sets property _group of the spans it returns
    //to the regexp group matched.  We can use this to determine if the token
    //is a word or punctuation.
```

```
    if (t._group == 1)
      token.tokenType = "Word";
    else
      token.tokenType = "Punctuation";
    emit(token);
  }
}

proc sentences(tokenStream)
{
  start = null;
  forAll (x: tokenStream)
  {
    if (start == null) start = x;
    if (x.tokenType == "Punctuation" &&
        matching("[.?!]",x))
    {
      emit (annotate ([start.begin, x.end],"Sentence"));
      start = null;
    }
  }
}

proc phoneNumbers(span)
{
  forAll(p: matching("(\(\d\d\d\)|\d\d\d-)\d\d\d-\d\d\d\d", span))
  {
    emit (annotate(p, "PhoneNumber"));
  }
}

proc phoneCalls(span)
{
   print("phoneCalls(");
   print(span);
   println(")");
   forAll(num1 : instancesOf("PhoneNumber",span))
   {
      println(num1);
      forAll (num2 : instancesOf("PhoneNumber",[num1.end,span.end]))
      {
        println(num2);
        if (matching("from",[span.begin, num1.begin]) &&
            matching("to",[num1.end, num2.begin]))
        {
          emit(annotate([span.begin, num2.end], "PhoneCall"));
        }
      }
    }
}

proc main(doc)
{
  forAll (s: sentences(tokens(doc)))
  {
    phoneNumbers(s);
    phoneCalls(s);
  }
}
```

The output of this program, when run on the sample document consisting of the single sentence "Intercept of a phone call from 321-555-4788 to 321-555-3391," is:

```
<?xml version="1.0" encoding="UTF-8"?>
<Document><Sentence><PhoneCall><Token>Intercept</Token>
<Token>of</Token> <Token>a</Token> <Token>phone</Token>
<Token>call</Token> <Token>from</Token>
<PhoneNumber>321-555-4788</PhoneNumber> <Token>to</Token>
<PhoneNumber>321-555-3391</PhoneNumber></PhoneCall>
<Token>.</Token></Sentence></Document>
```

Which is exactly the expected behavior. The program has marked the entire input text has a sentence, has successfully marked the individual words, and has located two PhoneNumbers and a PhoneCall between them.

7 Lessons Learned

The biggest lesson I learned was that climbing the learning curve of ANTLR was well worth the effort. The online user manual for ANTLR can be a bit daunting at first, but once you get the hang of it it's not so bad. I fully expect to get some use out of ANTLR at some point in the future.

My advice to future teams is to consider it a hard deadline to get your parser done at the same time the LRM is due, and then to employ an iterative development. More specifically:

- Start playing with ANTLR early, and build a very simple lexer and parser and make sure it works. Then go back and add features bit by bit, testing all the while. Repeat until you're happy with the feature set, then write that up in your LRM.

- Do the same for the back end. I started just implementing the `print` procedure and literals, and made sure that worked. Then I could implement simple arithmetic expressions, if statements, and so on, eventually getting to the more complicated things like procedure invocation and the more complex data types.

8 Code Listing

8.1 Overview

The following is a file listing for the entire Spaniel project, with each file preceded by its line count:

ANTLR Grammar Files

```
 241 grammar/spaniel.g
 219 grammar/walker.g
```

Package edu.columbia.apl2107.spaniel (Main Package)

```
  44 src/edu/columbia/apl2107/spaniel/AnnotatedDocument.java
 191 src/edu/columbia/apl2107/spaniel/Interpreter.java
  34 src/edu/columbia/apl2107/spaniel/JavaProcedure.java
```

Package edu.columbia.apl2107.spaniel.ast (Custom AST Classes)

```
 208 src/edu/columbia/apl2107/spaniel/ast/ArithmeticExpression.java
  34 src/edu/columbia/apl2107/spaniel/ast/Assignment.java
  16 src/edu/columbia/apl2107/spaniel/ast/AstNode.java
  75 src/edu/columbia/apl2107/spaniel/ast/BinOpExpression.java
  50 src/edu/columbia/apl2107/spaniel/ast/Block.java
  33 src/edu/columbia/apl2107/spaniel/ast/BreakStatement.java
  82 src/edu/columbia/apl2107/spaniel/ast/ConditionalExpression.java
  35 src/edu/columbia/apl2107/spaniel/ast/EmitStatement.java
  24 src/edu/columbia/apl2107/spaniel/ast/EmptyStatement.java
  57 src/edu/columbia/apl2107/spaniel/ast/Expression.java
  34 src/edu/columbia/apl2107/spaniel/ast/ExpressionStatement.java
  72 src/edu/columbia/apl2107/spaniel/ast/ExternalProcedure.java
  60 src/edu/columbia/apl2107/spaniel/ast/ForAllStatement.java
  62 src/edu/columbia/apl2107/spaniel/ast/IfStatement.java
  68 src/edu/columbia/apl2107/spaniel/ast/JavaCallExpression.java
  35 src/edu/columbia/apl2107/spaniel/ast/Literal.java
  92 src/edu/columbia/apl2107/spaniel/ast/LvalueExpression.java
  65 src/edu/columbia/apl2107/spaniel/ast/ProcCallExpression.java
 112 src/edu/columbia/apl2107/spaniel/ast/Procedure.java
 106 src/edu/columbia/apl2107/spaniel/ast/Program.java
 135 src/edu/columbia/apl2107/spaniel/ast/RelationalExpression.java
  42 src/edu/columbia/apl2107/spaniel/ast/SpanExpression.java
  21 src/edu/columbia/apl2107/spaniel/ast/Statement.java
  35 src/edu/columbia/apl2107/spaniel/ast/SymbolTable.java
 113 src/edu/columbia/apl2107/spaniel/ast/UnaryExpression.java
  46 src/edu/columbia/apl2107/spaniel/ast/VarEntry.java
  54 src/edu/columbia/apl2107/spaniel/ast/WhileStatement.java
```

Package edu.columbia.apl2107.spaniel.exception (Exception Classes)

```
  46 src/edu/columbia/apl2107/spaniel/exception/InternalException.java
  49 src/edu/columbia/apl2107/spaniel/exception/SpanielRuntimeException.java
  45 src/edu/columbia/apl2107/spaniel/exception/StaticSemanticsException.java
```

Package edu.columbia.apl2107.spaniel.parser (Lexer, Parser, and Tree Rewriter)
(NOTE: All files in this directory were generated by ANTLR)

```
1342 src/edu/columbia/apl2107/spaniel/parser/SpanielLexer.java
1557 src/edu/columbia/apl2107/spaniel/parser/SpanielParser.java
1058 src/edu/columbia/apl2107/spaniel/parser/SpanielTreeRewriter.java
  72 src/edu/columbia/apl2107/spaniel/parser/SpanielTreeRewriterTokenTypes.java
  72 src/edu/columbia/apl2107/spaniel/parser/SpanielVocabTokenTypes.java
```

Package edu.columbia.apl2107.procedures (Built-in Procedure Implementations)
```
  64 src/edu/columbia/apl2107/spaniel/procedures/Annotate.java
  53 src/edu/columbia/apl2107/spaniel/procedures/First.java
  64 src/edu/columbia/apl2107/spaniel/procedures/InstancesOf.java
  94 src/edu/columbia/apl2107/spaniel/procedures/Matching.java
  33 src/edu/columbia/apl2107/spaniel/procedures/Print.java
  33 src/edu/columbia/apl2107/spaniel/procedures/Println.java
  98 src/edu/columbia/apl2107/spaniel/procedures/REMatch.java
  54 src/edu/columbia/apl2107/spaniel/procedures/Rest.java
  63 src/edu/columbia/apl2107/spaniel/procedures/Subspans.java
```

Package edu.columbia.apl2107.spaniel.test (Test Cases)
```
  59 src/edu/columbia/apl2107/spaniel/test/JUnitAssert.java
  51 src/edu/columbia/apl2107/spaniel/test/JUnitAssertEquals.java
  74 src/edu/columbia/apl2107/spaniel/test/JUnitTestCase.java
  48 src/edu/columbia/apl2107/spaniel/test/TestParser.java
  48 src/edu/columbia/apl2107/spaniel/test/TestTreeRewriter.java
  46 src/edu/columbia/apl2107/spaniel/test/ViewAST.java
```

Package edu.columbia.apl2107.spaniel.util (Utility Classes)
```
  88 src/edu/columbia/apl2107/spaniel/util/ActivationRecord.java
  71 src/edu/columbia/apl2107/spaniel/util/AnnotatedDocumentImpl.java
  80 src/edu/columbia/apl2107/spaniel/util/InlineXmlOutput.java
  91 src/edu/columbia/apl2107/spaniel/util/UimaXmlOutput.java
```

Package edu.columbia.apl2107.spaniel.var_value (Variables, Values, and Types)
```
  52 src/edu/columbia/apl2107/spaniel/var_value/BooleanValue.java
  43 src/edu/columbia/apl2107/spaniel/var_value/FloatValue.java
  43 src/edu/columbia/apl2107/spaniel/var_value/IntegerValue.java
  38 src/edu/columbia/apl2107/spaniel/var_value/NullValue.java
  92 src/edu/columbia/apl2107/spaniel/var_value/SequenceValue.java
 207 src/edu/columbia/apl2107/spaniel/var_value/SpanValue.java
  45 src/edu/columbia/apl2107/spaniel/var_value/StringValue.java
  55 src/edu/columbia/apl2107/spaniel/var_value/Type.java
  50 src/edu/columbia/apl2107/spaniel/var_value/TypedVariable.java
  74 src/edu/columbia/apl2107/spaniel/var_value/Value.java
  63 src/edu/columbia/apl2107/spaniel/var_value/Variable.java
```

Spaniel Test Programs For JUnit Test Case
```
  21 test/junit/annotation.spl
  92 test/junit/controlFlow.spl
  34 test/junit/math.spl
  20 test/junit/recursion.spl
  68 test/junit/sequences.spl
  39 test/junit/spans.spl


  8984 total lines
 (4101 generated by ANTLR)
 =4883 lines written
```

8.2 ANTLR Grammar Files

8.2.1 spaniel.g

```
//*******************************************
//* Grammar file for the Spaniel language.
//* Includes Parser and Lexer.
//*
//* Author: alally
```

```
//*******************************************
header {package edu.columbia.apl2107.spaniel.parser;}

/*********************************
    PARSER
 *********************************/
class SpanielParser extends Parser;
options
{
  k = 2;
  buildAST = true;
  exportVocab = SpanielVocab;
}

tokens
{
  PROGRAM;
  FORMAL_PARAMS;
  BLOCK;
  PROC_CALL;
  JAVA_CALL;
  SPAN;
  EMPTY_STATEMENT;
  UPLUS;
  UMINUS;
}

/* Program is series of procecure declarations */
program
  : (procedure)+
    {#program = #([PROGRAM],program);}
  ;

procedure: "proc"^ ID formal_params  block;
formal_params:
    LPAREN! (ID (COMMA! ID)*)? RPAREN!
    {#formal_params = #([FORMAL_PARAMS],formal_params);};
block:
    LBRACE! (statement)* RBRACE!
    {#block = #([BLOCK],block);};

/* Statements */
statement
  : block
  | expression_statement
  | if_stmt
  | forAll_stmt
  | while_stmt
  | emit_stmt SEMI!
  | break_stmt SEMI!
  | SEMI! /* empty statement */
    {#statement = #([EMPTY_STATEMENT]);}
  ;

expression_statement
  : (assignment) => assignment SEMI!
  | proc_call_exp SEMI!;

if_stmt: "if"^ LPAREN! expression RPAREN! statement (options {greedy=true;}:"else"!
statement)?;

forAll_stmt: "forAll"^ LPAREN! ID COLON! expression RPAREN! statement;

while_stmt: "while"^ LPAREN! expression RPAREN! statement;

/* Procedures in Spaniel don't "return", they "emit" and keep running (at least
conceptually). */
emit_stmt: "emit"^ expression;

break_stmt: "break"^;
```

```
/* Expressions */

primary_exp
  : lvalue_exp
  | literal
  | proc_call_exp
  | span_exp
  | LPAREN! expression RPAREN!
  ;

/* For simplicity, Spaniel uses a relatively restricted form of the dot operator:
   It can only be appplied to identifiers.  For example, proc(x).field is not
   allowed as it would be in Java.  This turns out not to be a big issue
   because of the special nature of procedures in Spaniel (they return streams
   rather than objects). */
lvalue_exp: ID (DOT^ ID)*;

literal: StringLiteral^ | IntLiteral^ | FloatLiteral^ | "true"^ | "false"^ | "null"^;
proc_call_exp
  : ID actual_params
        {#proc_call_exp = #([PROC_CALL],proc_call_exp);}
  ;
actual_params: LPAREN! (expression (COMMA! expression)*)? RPAREN!;

/* Special Spaniel Syntax: [expression, expression] indicates a span definition */
span_exp
  : LBRACKET! expression COMMA! expression RBRACKET!
    {#span_exp = #([SPAN],span_exp);}
  ;

unary_exp
  : PLUS operand:unary_exp
    {#unary_exp = #([UPLUS],operand);}
  | MINUS operand2:unary_exp
    {#unary_exp = #([UMINUS],operand2);}
  | NOT^ unary_exp
  | primary_exp
  ;

mult_exp: unary_exp ((TIMES^ | DIV^ | MOD^) unary_exp )*;

add_exp: mult_exp ((PLUS^ | MINUS^) mult_exp)*;

rel_exp: add_exp ((EQ^ | NOTEQ^ | LT^ | LE^ | GT^ | GE^) add_exp)?;

and_exp: rel_exp (AND^ rel_exp)*;

or_exp: and_exp (OR^ and_exp)*;

/* Distinguish assignment expressions because they're valid statements. */
assign_exp
  : (lvalue_exp ASSIGN) => assignment
  | or_exp;
assignment: lvalue_exp ASSIGN^ assign_exp;

expression: assign_exp;


/********************************
    LEXER
 ********************************/

class SpanielLexer extends Lexer;
options
{
  k = 2;
  charVocabulary = '\3'..'\377';
  testLiterals = false;
  exportVocab = SpanielVocab;
}
```

```
/* Whitespace */
WS
  : ( ' ' | '\t' | LineTerminator )
    {$setType(Token.SKIP);};

/* LineTerminators: \r, \n, and \r \n (greedy match) */
protected LineTerminator
  : ('\r' (options {greedy=true;}:'\n')?  | '\n' )
    {newline();};
protected InputCharacter : ~('\r' | '\n');

/* Comments: both end of line (C++ style)and block (C style) comments supported */
Comment
  : (EndOfLineComment | BlockComment)
    {$setType(Token.SKIP);}
  ;
protected EndOfLineComment: '/' '/' (InputCharacter)* LineTerminator;
protected BlockComment: '/' '*' CommentTail;
protected CommentTail
  :     '*' CommentTailStar
  |     LineTerminator CommentTail
  |     ~('*' | '\r' | '\n') CommentTail
  ;
protected CommentTailStar
  :     '/' | '*' CommentTailStar
  | LineTerminator CommentTail
  | ~('*' | '/' | '\r' | '\n') CommentTail
  ;

/* Identifiers: letter or underscore followed by letters, digits, or underscores */
ID options {testLiterals = true;}: IdentifierStart (IdentifierChar)*;
protected IdentifierStart: '_' | 'A'..'Z' | 'a'..'z';
protected IdentifierChar: IdentifierStart | '0'..'9';

/* Numeric Literals -- the lexer distinguishes between integer and floating-point
literals.
   This is a little tricky to do since they share a common prefix.  We use a single rule
   to match both, using the $setType command to assign the appropriate type.  */
IntOrFloatLiteral:
        IntLiteral (FloatTail {$setType(FloatLiteral);} | /* nothing */
{$setType(IntLiteral);});
/* Also match floating-point literals with no integer part, using a separate rule. */
FloatLiteral: Fraction (Exponent)? {$setType(FloatLiteral);};
protected IntLiteral: '0' | '1'..'9' (Digit)*;
protected FloatTail: (Fraction (Exponent)? | Exponent);
protected Digit: ('0'..'9');
protected Digits: (Digit)+;
protected Fraction: '.' Digits;
protected Exponent: 'e' ('+'|'-')? Digits;

/* String Literals -- two forms:
   (1) A sequence of characters enclosed in double quotes (").  We use repeated double
   quotes ("") to represent an embedded quote.  This is done to make it easier to write
   string literals containing backslashes, such as regular expressions.
   (2) A sequence of (# symbol followed by one or two hex digits).  This is how
   special characters such as newlines get represented. */
StringLiteral
  : '"'! (~('"' | '\n' | '\r') | '"'! '"')* '"'!
  | (CharacterCode)+;
protected CharacterCode
  : '#'! HexDigit (HexDigit)?
    {
      //set the text of this rule to the character having this ASCII value
      char c = (char)Integer.parseInt($getText,16);
      $setText(c);
    }
  ;
protected HexDigit: (Digit | 'A'..'F' | 'a'..'f');

/* Separators */
```

```
LPAREN: '(';
RPAREN: ')';
LBRACKET: '[';
RBRACKET: ']';
LBRACE: '{';
RBRACE: '}';
COMMA: ',';
SEMI: ';';
DOT: '.';
COLON: ':';

/* Operators */
ASSIGN : '=';
EQ : '=' '=';
NOTEQ : '!' '=';
LT: '<';
LE: '<' '=';
GT: '>';
GE: '>' '=';
PLUS: '+';
MINUS: '-';
TIMES: '*';
DIV: '/';
MOD: '%';
AND: '&' '&';
OR : '|' '|';
NOT : '!';
```

8.2.2 walker.g

```
//*****************************************
//* Grammar file for the Spaniel AST walker.
//* Translates the ANTLR homogeous AST into a custom heterogeneous AST,
//* while checking static semantics.
//*
//* Author: alally
//*****************************************
header {package edu.columbia.apl2107.spaniel.parser;}

{
import java.io.*;
import java.util.*;
import edu.columbia.apl2107.spaniel.ast.*;
import edu.columbia.apl2107.spaniel.var_value.*;
import edu.columbia.apl2107.spaniel.exception.*;
}

class SpanielTreeRewriter extends TreeParser;
options
{
  importVocab = SpanielVocab;
}


program returns [ Program prog ]
{
  prog = new Program();
  Procedure p;
  List procBodies = new ArrayList();
  List params;
}
  : #(PROGRAM (#("proc" name:ID params=formal_params (body:.)?)
      {
        //create new SymbolTable for the procedure
        SymbolTable symTab = new SymbolTable(prog.getSymbolTable());

        //create Procedure object.  Note this populates symTab with formal params
            Procedure proc = new Procedure(name.getText(), params, symTab);
            //add to program (this populates global symbol table with proc name)
        prog.addProcedure(proc);
        //Store body AST node on list so we can parse them in a second pass
```

```
        procBodies.add(body);
      }
     )*)
{
  //now parse bodies
  Iterator procIterator = prog.getProcedures().iterator();
  Iterator bodyIterator = procBodies.iterator();
  while(procIterator.hasNext())
  {
    Procedure proc = (Procedure)procIterator.next();
    body = (AST)bodyIterator.next();
    proc.setBody(block(#body,proc.getSymbolTable(),false));
  }
};

formal_params returns [List r]
{
  r = new ArrayList();
}
  : #(FORMAL_PARAMS (param:ID {r.add(param.getText());})*);

block [SymbolTable symTab, boolean inLoop] returns [Block r]
{
  r = null;
  List statements = new ArrayList();
  Statement s;
}
        : #(BLOCK (s=statement[symTab,inLoop] {statements.add(s);})*)
{
  r = new Block(statements);
};

statement [SymbolTable symTab, boolean inLoop]  returns [Statement r]
{
  r = null;
  Expression e;
  Statement s = null, thenS = null, elseS = null;
  LvalueExpression l;
}
  : #("if" e=expr[symTab] thenS=statement[symTab,inLoop]
(elseS=statement[symTab,inLoop])?)
        { r = new IfStatement(e,thenS,elseS);}
  | #("forAll" l=lvalue_exp[symTab] e=expr[symTab] s=statement[symTab,true])
    { r = new ForAllStatement(l,e,s);}
  | #("while" e=expr[symTab] s=statement[symTab,true])
    { r = new WhileStatement(e,s);}
  | #("emit" e=expr[symTab])
    { r = new EmitStatement(e); }
  | "break"
    { //must be in loop
      if (!inLoop)
        throw new StaticSemanticsException("break occurred outside of loop");
      r = new BreakStatement();
        }
  | e=expr[symTab] {r = new ExpressionStatement(e);}
  | r=block[symTab,inLoop]
  | EMPTY_STATEMENT {r = new EmptyStatement();}
  ;


expr [SymbolTable symTab] returns [Expression r]
{
  Expression a,b;
  r = null;
}
  : #(ASSIGN a=expr[symTab] b=expr[symTab])
    { r = new Assignment(a,b); }
  | #(EQ a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(EQ,a,b); }
  | #(GE a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(GE,a,b); }
```

```
  | #(LE a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(LE,a,b); }
  | #(GT a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(GT,a,b); }
  | #(LT a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(LT,a,b); }
  | #(NOTEQ a=expr[symTab] b=expr[symTab])
    { r = new RelationalExpression(NOTEQ,a,b); }
  | #(PLUS a=expr[symTab] b=expr[symTab])
    { r = new ArithmeticExpression(PLUS,a,b); }
  | #(MINUS a=expr[symTab] b=expr[symTab])
    { r = new ArithmeticExpression(MINUS,a,b); }
  | #(TIMES a=expr[symTab] b=expr[symTab])
    { r = new ArithmeticExpression(TIMES,a,b); }
  | #(DIV a=expr[symTab] b=expr[symTab])
    { r = new ArithmeticExpression(DIV,a,b); }
  | #(MOD a=expr[symTab] b=expr[symTab])
    { r = new ArithmeticExpression(MOD,a,b); }
  | #(AND a=expr[symTab] b=expr[symTab])
    { r = new ConditionalExpression(AND,a,b); }
  | #(OR a=expr[symTab] b=expr[symTab])
    { r = new ConditionalExpression(OR,a,b); }
  | #(UPLUS a=expr[symTab])
    { r = new UnaryExpression(UPLUS,a); }
  | #(UMINUS a=expr[symTab])
    { r = new UnaryExpression(UMINUS,a); }
  | #(NOT a=expr[symTab])
    { r = new UnaryExpression(NOT,a); }
  | s:StringLiteral
    { r = new Literal(new StringValue(s.getText()));}
  | i:IntLiteral
    { r = new Literal(new IntegerValue(Integer.parseInt(i.getText())));}
  | f:FloatLiteral
    { r = new Literal(new FloatValue(Float.parseFloat(f.getText())));}
  | "true"
    { r = new Literal(BooleanValue.TRUE);}
  | "false"
    { r = new Literal(BooleanValue.FALSE);}
  | "null"
    { r = new Literal(NullValue.instance());}
  | #(SPAN a=expr[symTab] b=expr[symTab])
    {r = new SpanExpression(a,b);}
  | r=lvalue_exp[symTab]
  | r=proc_call_exp[symTab]
  ;

/* For lvalue expressions, bind names to Variable entries in the synbol table. */
lvalue_exp [SymbolTable symTab] returns [LvalueExpression r]
{
  r = null;
  LvalueExpression lhs;
}
  : #(DOT lhs=lvalue_exp[symTab] rhs:ID)
    {r = new LvalueExpression(lhs,rhs.getText());}
  | n:ID
    { //resolve this name to a variable using the symbol table
      Object symbol = symTab.get(n.getText());
      if (symbol == null)
      {
        //create new VarEntry
        symbol = new VarEntry(n.getText(), symTab.size());
        symTab.put(n.getText(), symbol);
      }
      assert(symbol instanceof VarEntry);
      r = new LvalueExpression((VarEntry)symbol);
    }
  ;

/* For proc call expressions, bind names to Procedure entries in the synbol table. */
proc_call_exp [SymbolTable symTab] returns [Expression r]
{
```

```
  r = null;
  List params = new ArrayList();
  Expression e;
}
    : #(PROC_CALL pname:ID (e=expr[symTab] {params.add(e);})*)
{
  //The "javacall" procedure is a special case
  if ("javacall".equals(pname.getText()))
  {
    //must have at least one parameter
    if (params.isEmpty())
    {
      throw new StaticSemanticsException("javacall must have at least one argument");
    }
    r = new JavaCallExpression(params);
  }
  else
  {
    //resolve the proc name to a procedure using the symbol table
    //we know all procedures are defined in the parent (global) symbol table
    Object symbol = symTab.getParent().get(pname.getText());
    if (symbol == null)
    {
      throw new StaticSemanticsException("Call to undefined procedure " +
pname.getText());
    }
    assert (symbol instanceof Procedure);
    Procedure proc = (Procedure)symbol;
    //check arity
    int arity = proc.getFormalParams().size();
    if (params.size() != arity )
    {
      throw new StaticSemanticsException("Procedure " + pname.getText() + " requires " +
arity + " arguments.");
    }
    r = new ProcCallExpression(proc, params);
  }
};
```

8.3 *Package edu.columbia.apl2107.spaniel*

8.3.1 ArithmeticExpression.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.FloatValue;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.NullValue;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for binary arithmetic (+,-,*,/,%) expressions .
 * @author alally
 */
public class ArithmeticExpression extends BinOpExpression
{
  public ArithmeticExpression(int op, Expression lhs, Expression rhs)
  {
    super(op, lhs, rhs);
  }
```

```java
/* (non-Javadoc)
 * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
 */
public Object eval(ActivationRecord activationRecord)
{
  Value v1 = getLhs().getValue(activationRecord).convertSequence();
  Value v2 = getRhs().getValue(activationRecord).convertSequence();

  switch(getOperator())
  {
    case SpanielVocabTokenTypes.PLUS:
      if (v1 instanceof IntegerValue)
      {
        int int1 = ((IntegerValue)v1).getInt();
        if (v2 instanceof IntegerValue)
        {
          return new IntegerValue(int1 + ((IntegerValue)v2).getInt());
        }
        else if (v2 instanceof FloatValue)
        {
          return new FloatValue(int1 + ((FloatValue)v2).getFloat());
        }
      }
      else if (v1 instanceof FloatValue)
      {
        float float1 = ((FloatValue)v1).getFloat();
        if (v2 instanceof IntegerValue)
        {
          return new FloatValue(float1 + ((IntegerValue)v2).getInt());
        }
        else if (v2 instanceof FloatValue)
        {
          return new FloatValue(float1 + ((FloatValue)v2).getFloat());
        }
      }
      else if (v1 instanceof StringValue && v2 instanceof StringValue)
      {
        return new StringValue(
            ((StringValue)v1).getString() + ((StringValue)v2).getString());
      }
      else if (v1 instanceof StringValue && v2 instanceof IntegerValue)
      {
        return new StringValue(
            ((StringValue)v1).getString() + ((IntegerValue)v2).getInt());
      }
      else if (v1 instanceof SpanValue && v2 instanceof SpanValue)
      {
        //+ operator on spans computes minimal enclosing span
        SpanValue s1 = (SpanValue)v1;
        SpanValue s2 = (SpanValue)v2;
        int b1 = s1.getBegin().getInt();
        int e1 = s1.getEnd().getInt();
        int b2 = s2.getBegin().getInt();
        int e2 = s2.getEnd().getInt();
        return new SpanValue(
            new IntegerValue(Math.min(b1,b2)),
            new IntegerValue(Math.max(e1,e2)));
      }
      //fall-through if invalid types
      throw new SpanielRuntimeException("Invalid types to + operator: " +
          v1.getTypeName() + "," + v2.getTypeName());

    case SpanielVocabTokenTypes.MINUS:
      if (v1 instanceof IntegerValue)
      {
        int int1 = ((IntegerValue)v1).getInt();
        if (v2 instanceof IntegerValue)
        {
          return new IntegerValue(int1 - ((IntegerValue)v2).getInt());
        }
```

```
      else if (v2 instanceof FloatValue)
      {
        return new FloatValue(int1 - ((FloatValue)v2).getFloat());
      }
    }
    else if (v1 instanceof FloatValue)
    {
      float float1 = ((FloatValue)v1).getFloat();
      if (v2 instanceof IntegerValue)
      {
        return new FloatValue(float1 - ((IntegerValue)v2).getInt());
      }
      else if (v2 instanceof FloatValue)
      {
        return new FloatValue(float1 - ((FloatValue)v2).getFloat());
      }
    }
    //fall-through if invalid types
    throw new SpanielRuntimeException("Invalid types to - operator: " +
        v1.getTypeName() + "," + v2.getTypeName());

  case SpanielVocabTokenTypes.TIMES:
    if (v1 instanceof IntegerValue)
    {
      int int1 = ((IntegerValue)v1).getInt();
      if (v2 instanceof IntegerValue)
      {
        return new IntegerValue(int1 * ((IntegerValue)v2).getInt());
      }
      else if (v2 instanceof FloatValue)
      {
        return new FloatValue(int1 * ((FloatValue)v2).getFloat());
      }
    }
    else if (v1 instanceof FloatValue)
    {
      float float1 = ((FloatValue)v1).getFloat();
      if (v2 instanceof IntegerValue)
      {
        return new FloatValue(float1 * ((IntegerValue)v2).getInt());
      }
      else if (v2 instanceof FloatValue)
      {
        return new FloatValue(float1 * ((FloatValue)v2).getFloat());
      }
    }
    else if (v1 instanceof SpanValue && v2 instanceof SpanValue)
    {
      //* operator on spans computes intersection
      SpanValue s1 = (SpanValue)v1;
      SpanValue s2 = (SpanValue)v2;
      int b1 = s1.getBegin().getInt();
      int e1 = s1.getEnd().getInt();
      int b2 = s2.getBegin().getInt();
      int e2 = s2.getEnd().getInt();
      if (b2 > e1 || b1 > e2)
        return NullValue.instance();
      else
        return new SpanValue(
          new IntegerValue(Math.max(b1,b2)),
          new IntegerValue(Math.min(e1,e2)));
    }
    //fall-through if invalid types
    throw new SpanielRuntimeException("Invalid types to * operator: " +
          v1.getTypeName() + "," + v2.getTypeName());

  case SpanielVocabTokenTypes.DIV:
    if (v1 instanceof IntegerValue)
    {
      int int1 = ((IntegerValue)v1).getInt();
      if (v2 instanceof IntegerValue)
```

```
          {
            return new IntegerValue(int1 / ((IntegerValue)v2).getInt());
          }
          else if (v2 instanceof FloatValue)
          {
            return new FloatValue(int1 / ((FloatValue)v2).getFloat());
          }
        }
        else if (v1 instanceof FloatValue)
        {
          float float1 = ((FloatValue)v1).getFloat();
          if (v2 instanceof IntegerValue)
          {
            return new FloatValue(float1 / ((IntegerValue)v2).getInt());
          }
          else if (v2 instanceof FloatValue)
          {
            return new FloatValue(float1 / ((FloatValue)v2).getFloat());
          }
        }
        //fall-through if invalid types
        throw new SpanielRuntimeException("Invalid types to / operator: " +
              v1.getTypeName() + "," + v2.getTypeName());

      case SpanielVocabTokenTypes.MOD:
        if (v1 instanceof IntegerValue && v2 instanceof IntegerValue)
        {
          return new IntegerValue(
           ((IntegerValue)v1).getInt() % ((IntegerValue)v2).getInt());
        }
        else
          throw new SpanielRuntimeException("Invalid types to % operator: " +
              v1.getTypeName() + "," + v1.getTypeName());

      default:
        assert false : "Unknown operator";
        return null;
    }
  }

}
```

8.3.2 Assignment.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Custom AST node class for assignment (=) expressions.
 * @author alally
 */
public class Assignment extends BinOpExpression
{
  public Assignment(Expression lhs, Expression rhs)
  {
    super(SpanielVocabTokenTypes.ASSIGN, lhs, rhs);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    Object lhsVar = getLhs().eval(activationRecord);
```

```
    assert lhsVar instanceof Variable;
    Value val = getRhs().getValue(activationRecord);
    ((Variable)lhsVar).setValue(val);
    return val;
  }

}
```

8.3.3 AstNode.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

/**
 * The base class for Spaniel custom AST nodes.  There is a subclass of AstNode for each
 * of the expression and statement types defined in the LRM (with the exception that
 * the various types of binary expressions are collapsed into the three classes
 * ArithmeticExpression, ConditionalExpression, and RelationalExpression).
 *
 * @author alally
 */
public abstract class AstNode
{

}
```

8.3.4 BinOpExpression.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;

/**
 * Custom AST Node abstract class for expressions consisting of a binary operator and
 * two operands.  Extended by concrete classes that implement evaluation logic for
 * the different types of binary expressions.
 *
 * @author alally
 */
public abstract class BinOpExpression extends Expression
{
  private int _operator;
  private Expression _lhs;
  private Expression _rhs;

  /**
   * Create a new BinOpExpression.
   *
   * @param op the operator.  The integer code used here is that generated by
   *   ANLTR in the class {@link SpanielVocabTokenTypes}.
   * @param lhs the left-hand-side expression
   * @param rhs the right-hand-side expression
   */
  public BinOpExpression(int op, Expression lhs, Expression rhs)
  {
    _operator = op;
    _lhs = lhs;
    _rhs = rhs;
  }

  public String toString()
  {
    return "(" + getOperatorString() + " " + _lhs + " " + _rhs + ")";
  }
```

```
  public String getOperatorString()
  {
    switch (_operator)
    {
      case SpanielVocabTokenTypes.PLUS: return "+";
      case SpanielVocabTokenTypes.MINUS: return "-";
      case SpanielVocabTokenTypes.TIMES: return "*";
      case SpanielVocabTokenTypes.DIV: return "/";
      case SpanielVocabTokenTypes.MOD: return "/";
      case SpanielVocabTokenTypes.EQ: return "==";
      case SpanielVocabTokenTypes.NOTEQ: return "!=";
      case SpanielVocabTokenTypes.LT: return "<";
      case SpanielVocabTokenTypes.LE: return "<=";
      case SpanielVocabTokenTypes.GT: return ">";
      case SpanielVocabTokenTypes.GE: return ">=";
      case SpanielVocabTokenTypes.ASSIGN: return "=";
      default: return null;
    }
  }

  protected int getOperator()
  {
    return _operator;
  }

  protected Expression getLhs()
  {
    return _lhs;
  }

  protected Expression getRhs()
  {
    return _rhs;
  }
}
```

8.3.5 Block.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.Iterator;
import java.util.List;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node class for a block, which is a series of {@link Statement}s.
 * @author alally
 */
public class Block extends Statement
{
  private List _statements;

  public Block(List statements)
  {
    _statements = statements;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    Iterator it = _statements.iterator();
    while (it.hasNext() && !activationRecord.isBreakFlag())
    {
      ((Statement)it.next()).execute(activationRecord);
    }
```

```
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    Iterator it = _statements.iterator();
    while (it.hasNext())
    {
      Statement s = (Statement) it.next();
      if (s != null)
       buf.append(s.toString()).append('\n');
    }
    return buf.toString();
  }

}
```

8.3.6 BreakStatement.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node class for a break statement.  Break is implemented by setting a flag
 * in the current activation record.  Loop statements and Blocks check this flag and
 * terminate if it is set.
 *
 * @author alally
 */
public class BreakStatement extends Statement
{
  public BreakStatement()
  {
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    activationRecord.setBreakFlag(true);
  }

  public String toString()
  {
    return "(break)";
  }
}
```

8.3.7 ConditionalExpression.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for a conditional (&&, ||) expression.
 *
 * @author alally
```

```
 */
public class ConditionalExpression extends BinOpExpression
{
  public ConditionalExpression(int op, Expression lhs, Expression rhs)
  {
    super(op, lhs, rhs);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval()
   */
  public Object eval(ActivationRecord activationRecord)
  {
    Value v1, v2;
    switch(getOperator())
    {
      case SpanielVocabTokenTypes.AND:
        v1 = getLhs().getValue(activationRecord).coerceToBoolean();
        if (!(v1 instanceof BooleanValue))
        {
          throw new SpanielRuntimeException("Invalid type to && operator: " +
              v1.getTypeName());
        }
        if (!((BooleanValue)v1).getBoolean())
        {
          return v1; //short-circuit, return false
        }
        v2 = getRhs().getValue(activationRecord).coerceToBoolean();
        if (!(v2 instanceof BooleanValue))
        {
          throw new SpanielRuntimeException("Invalid type to && operator: " +
              v2.getTypeName());
        }
        else
        {
          return v2;
        }

      case SpanielVocabTokenTypes.OR:
        v1 = getLhs().getValue(activationRecord).coerceToBoolean();
        if (!(v1 instanceof BooleanValue))
        {
          throw new SpanielRuntimeException("Invalid type to || operator: " +
              v1.getTypeName());
        }
        if (((BooleanValue)v1).getBoolean())
        {
          return v1; //short-circuit, return true
        }
        v2 = getRhs().getValue(activationRecord).coerceToBoolean();
        if (!(v2 instanceof BooleanValue))
        {
          throw new SpanielRuntimeException("Invalid type to || operator: " +
              v2.getTypeName());
        }
        else
        {
          return v2;
        }

      default:
        assert false : "Unknown operator";
        return null;
    }
  }

}
```

8.3.8 EmitStatement.java

```
/*
```

```
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node class for an emit statement.  Adds a value to the current procedure's
 * return sequence, which is stored in the activation record.
 *
 * @author alally
 */
public class EmitStatement extends Statement
{
  private Expression _expr;

  public EmitStatement(Expression expr)
  {
    _expr = expr;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute()
   */
  public void execute(ActivationRecord activationRecord)
  {
    activationRecord.getReturnValue().append(_expr.getValue(activationRecord));
  }

  public String toString()
  {
    return "(emit" + _expr.toString() + ")";
  }
}
```

8.3.9 EmptyStatement.java

```
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node class for an empty statement.
 *
 * @author alally
 */
public class EmptyStatement extends Statement
{

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    //does nothing
  }

}
```

8.3.10 Expression.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;
```

```java
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Custom AST node abstract class for an expression.  Subclassed by the various
 * types of expressions.
 *
 * @author alally
 */
public abstract class Expression extends AstNode
{
  /**
   * Evaluate to either a Variable or a Value.
   *
   * @param activationRecord the activation record for the currently executing procedure
   *
   * @return the Variable or Value to which this expression evaluates.
   */
  public abstract Object eval(ActivationRecord activationRecord);

  /**
   * Evaluate to a value.  (If this expression evaluates to a Variable,
   * return the value of that variable.)
   *
   * @param activationRecord the activation record for the currently executing procedure
   *
   * @return the value to which this expression evaluates
   */
  public Value getValue(ActivationRecord activationRecord)
  {
    Object v = eval(activationRecord);
    if (v instanceof Value)
    {
      return (Value)v;
    }
    else if (v instanceof Variable)
    {
      return ((Variable)v).getValue();
    }
    else if (v == null)
    {
      return null;
    }
    else
    {
      assert false : "Expression must evaluate to a variable or a value";
      return null;
    }

  }

}
```

8.3.11 ExpressionStatement.java

```java
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node class for an expression statement.
 *
 * @author alally
 */
public class ExpressionStatement extends Statement
{
  Expression _expr;
```

```
  public ExpressionStatement(Expression expr)
  {
    _expr = expr;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    _expr.eval(activationRecord);
  }

   public String toString()
   {
     return _expr.toString();
   }
}
```

8.3.12 ExternalProcedure.java

```
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.List;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.InternalException;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * A subtype of Procedure representing procedures that aren't defined in the
 * user's program.  This includes built-in procedures (which appear in the
 * global symbol table so that procedure call expressions can refer to them)
 * and user-defined procedures that are referenced by <code>javacall</code>
 * expressions in the user's program.
 *
 * @author alally
 */
public class ExternalProcedure extends Procedure
{

  private Class _implementingClass;

  /**
   * @param name
   * @param formalParams
   * @param symbolTable
   */
  public ExternalProcedure(String name, List formalParams,
      Class implementingClass)
  {
    super(name, formalParams, new SymbolTable());
    assert implementingClass != null;
    _implementingClass = implementingClass;
  }

  /*
   * (non-Javadoc)
   *
   * @see edu.columbia.apl2107.spaniel.ast.Procedure#execute(AnnotatedDocument,Value[])
   */
  public SequenceValue invoke(AnnotatedDocument doc, Value[] args)
  {
    //create a new instanceof the class implementing the procedure
    JavaProcedure javaProc;
    try
```

```java
    {
      javaProc = (JavaProcedure) _implementingClass.newInstance();
    }
    catch (InstantiationException e)
    {
      throw new InternalException(e);
    } catch (IllegalAccessException e)
    {
      throw new InternalException(e);
    }
    //initialize
    javaProc.init(doc,args);
    //call next() repeatedly and build SequenceValue for return
    SequenceValue returnVal = new SequenceValue();
    Value v = javaProc.next();
    while(v != null)
    {
      returnVal.append(v);
      v = javaProc.next();
    }
    return returnVal;
  }
}
```

8.3.13 ForAllStatement.java

```java
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Type;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Custom AST node class for a forAll statement.
 *
 * @author alally
 */
public class ForAllStatement extends Statement
{
  private LvalueExpression _var;
  private Expression _expr;
  private Statement _body;

  public ForAllStatement(LvalueExpression var, Expression expr, Statement body)
  {
    _var = var;
    _expr = expr;
    _body = body;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    Variable loopVar = (Variable)_var.eval(activationRecord);
    Value exprVal = _expr.getValue(activationRecord);
    if (exprVal.getType() != Type.SEQUENCE)
    {
      throw new SpanielRuntimeException("forAll statement expression must evaluate to a
sequence");
    }
    SequenceValue sequence = (SequenceValue)exprVal;
    while (!sequence.isEmpty() && !activationRecord.isBreakFlag())
    {
```

```
      //assign head of sequence to loop varible
      loopVar.setValue(sequence.getHead());
      //execute body
      _body.execute(activationRecord);
      //repeat on tail of sequence
      sequence = sequence.getTail();
    }
    activationRecord.setBreakFlag(false);
  }

  public String toString()
  {
    return "(forAll " + _var.toString() + " " + _expr.toString() + " " + _body.toString()
+ ")";

  }
}
```

8.3.14 IfStatement.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for an if statement.
 * @author alally
 */
public class IfStatement extends Statement
{
  private Expression _condition;
  private Statement _thenStatement;
  private Statement _elseStatement;

  public IfStatement(Expression condition, Statement thenStatement, Statement
elseStatement)
  {
    _condition = condition;
    _thenStatement = thenStatement;
    _elseStatement = elseStatement;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute()
   */
  public void execute(ActivationRecord activationRecord)
  {
    Value conditionVal =
        _condition.getValue(activationRecord).coerceToBoolean();
    if (conditionVal instanceof BooleanValue)
    {
      if (((BooleanValue)conditionVal).getBoolean())
      {
        _thenStatement.execute(activationRecord);
      }
      else
      {
        if (_elseStatement != null)
        {
          _elseStatement.execute(activationRecord);
        }
      }
    }
    else
    {
```

```
      throw new SpanielRuntimeException(
          "Non-boolean value " + conditionVal + " in if statement condition");
    }

  }

  public String toString()
  {
    return "(if " + _condition.toString() + " " + _thenStatement.toString() +
           (_elseStatement != null ? _elseStatement.toString() : "") + ")";
  }
}
```

8.3.15 JavaCallExpression.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import edu.columbia.apl2107.spaniel.exception.StaticSemanticsException;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for a JavaCall statement.  This is implemented as a type of
 * Procedure Call Expression.  It has different constructor logic (using Java reflection
 * to locate a Java class that implements the procedure, rather than using the symbol
 * table), but at runtime its behavior is identical to ProcCallExpression.
 * @author alally
 */
public class JavaCallExpression extends ProcCallExpression
{
  public JavaCallExpression(List params)
  {
    super(null,null);
    //first param must be StringLiteral -- the class name
    assert !params.isEmpty();
    if (params.get(0) instanceof Literal)
    {
      Value className = ((Literal)params.get(0)).getValue(null);
      if (className instanceof StringValue)
      {
        Class implementingClass;
        try
        {
          implementingClass = Class.forName(((StringValue)className).getString());
        }
        catch (ClassNotFoundException e)
        {
          throw new StaticSemanticsException("javacall class not found: " +
e.getMessage());
        }
        _proc = new ExternalProcedure(implementingClass.getName(), new ArrayList(),
implementingClass);
        //save the rest of the argument expressions for later
        _params = new ArrayList(params);
        _params.remove(0);
      }
      else
        throw new StaticSemanticsException("First argument to javacall must be a
StringLiteral");
    }
    else
      throw new StaticSemanticsException("First argument to javacall must be a
StringLiteral");
```

```
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    buf.append("(javacall ").append(_proc.getName());
    Iterator it = _params.iterator();
    while (it.hasNext())
    {
      buf.append(" ");
      buf.append(it.next().toString());
    }
    buf.append(')');
    return buf.toString();
  }
}
```

8.3.16 Literal.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for any type of literal.
 *
 * @author alally
 */
public class Literal extends Expression
{
  Value _value;

  public Literal(Value value)
  {
    _value = value;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval()
   */
  public Object eval(ActivationRecord activationRecord)
  {
    return _value;
  }

  public String toString()
  {
    return _value.toString();
  }
}
```

8.3.17 LvalueExpression.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Custom AST node class for Lvalue Expressions, which evaluate to Variables.
```

```
 *
 * @author alally
 */
public class LvalueExpression extends Expression
{
  LvalueExpression _lhsExp;
  VarEntry _lhsVar;
  String _rhsId;

  /**
   * Creates a new Lvalue expression of the form: <br/>
   * <i>lvalue_exp</li><code>.</code><i>identifier</i><br/>
   *
   * @param lhs the Lvalue expression on the left-hand-side of the dot
   * @param rhs the identifier on the right-hand-side of the dot
   */
  public LvalueExpression(LvalueExpression lhs, String rhs)
  {
    _lhsExp = lhs;
    _rhsId = rhs;
  }

  /**
   * Creates a new Lvalue expression consisting of a single identifier.
   *
   * @param var the VarEntry in the symbol table for the identifier
   */
  public LvalueExpression(VarEntry var)
  {
    _lhsVar = var;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    if (_lhsVar != null)
    {
      //get the local Variable from the current activation record
      return activationRecord.getLocalVariable(_lhsVar.getIndex());
    }
    else
    {
      //evaluate lhsExp to a Span value, and get field variable from it
      Value lhsVal = _lhsExp.getValue(activationRecord).convertSequence();
      if (lhsVal instanceof SpanValue)
      {
        SpanValue span = (SpanValue)lhsVal;
        Variable fieldVar = span.getField(_rhsId);
        //if field doesn't exist, create it with null value
        if (fieldVar == null)
        {
          fieldVar = span.createField(_rhsId);
        }
        return fieldVar;
      }
      else
      {
        throw new SpanielRuntimeException("Field access attempted on value of type " +
lhsVal.getTypeName());
      }
    }

  }

  public String toString()
  {
    if (_lhsExp != null)
    {
      return _lhsExp.toString() + "." + _rhsId;
```

```
    }
    else
    {
      return _lhsVar.toString();
    }

  }

}
```

8.3.18 ProcCallExpression.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.Iterator;
import java.util.List;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for Procedure Call Expressions.
 *
 * @author alally
 */
public class ProcCallExpression extends Expression
{
  protected Procedure _proc;
  protected List _params;

  /**
   * Creates a new Procedure Call Expression.
   *
   * @param proc reference to the procedure to be called.  This could be a procedure
   *   defined in the user's program or an {@link ExternalProcedure}.
   * @param params a list of {@link Expression} objects that are evaluated to produce
   *   the actual parameters to be passed to the procedure.
   */
  public ProcCallExpression(Procedure proc, List params)
  {
    _proc = proc;
    _params = params;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    Value[] actualParams = new Value[_params.size()];
    for (int i = 0; i < actualParams.length; i++)
    {
      actualParams[i] = ((Expression)_params.get(i)).getValue(activationRecord);
    }

    //evaluate parameters
    return _proc.invoke(activationRecord.getAnnotatedDocument(), actualParams);
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append(_proc.getName());
    Iterator it = _params.iterator();
    while (it.hasNext())
    {
      buf.append(" ");
```

```
      buf.append(it.next().toString());
    }
    buf.append(")");
    return buf.toString();
  }
}
```

8.3.19 Procedure.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node for a procedure.  Direct instances of this class represent user-
defined
 * procedures.  However, there is also the subclass {@link ExternalProcedure}, which
 * represents procedures implemented in Java that can be called from a Spaniel program.
 *
 * @author alally
 */
public class Procedure extends AstNode
{
  private String _name;

  private List _formalParams;

  private Block _body;

  /** Symbol table that stores local variables for this procedure. */
  private SymbolTable _symbolTable = new SymbolTable();

  /**
   * Create a new Procedure.
   *
   * @param name name of the procedure
   * @param formalParams list of Strings that are the names of the formal parameters
   *   to this Procedure.  This method will create VarEntries in the SymbolTable for
   *   all of these parameters.
   * @param symbolTable the symbol table that will hold the parameter variables and
   *   local variables for this procedure.
   */
  public Procedure(String name, List formalParams, SymbolTable symbolTable)
  {
    _name = name;
    _symbolTable = symbolTable;
    _formalParams = new ArrayList();
    //create VarEntries in symbol table for the formal params
    for (int i = 0; i < formalParams.size(); i++)
    {
      String paramName = (String)formalParams.get(i);
      VarEntry entry = new VarEntry(paramName, i);
      _symbolTable.put(paramName, entry);
      _formalParams.add(entry);
    }
  }

  /**
   * Invoke this procedure.
   *
```

```
   * @param doc the AnnotatedDocument on which the procedure should operate
   * @param args the arguments to the procdedure
   * @return the value returned by the procedure (always a sequence)
   */
  public SequenceValue invoke(AnnotatedDocument doc, Value[] args)
  {
    //create new activation record and execute procedure body
    ActivationRecord activationRecord = new ActivationRecord(this, doc, args);
    _body.execute(activationRecord);
    return activationRecord.getReturnValue();
  }

  public String getName()
  {
    return _name;
  }

  public List getFormalParams()
  {
    return Collections.unmodifiableList(_formalParams);
  }

  public SymbolTable getSymbolTable()
  {
    return _symbolTable;
  }

  public void setBody(Block block)
  {
    _body = block;
  }

  public String toString()

  {
    StringBuffer buf = new StringBuffer();
    buf.append("proc " + _name + "(");
    Iterator it = _formalParams.iterator();
    while (it.hasNext())
    {
      buf.append(it.next());
      if (it.hasNext())
        buf.append(',');
    }
    buf.append(")");
    if (_body != null)
    {
      buf.append('\n').append(_body.toString());
    }
    return buf.toString();
  }
}
```

8.3.20 Program.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.exception.StaticSemanticsException;
import edu.columbia.apl2107.spaniel.procedures.Annotate;
import edu.columbia.apl2107.spaniel.procedures.First;
import edu.columbia.apl2107.spaniel.procedures.InstancesOf;
import edu.columbia.apl2107.spaniel.procedures.Matching;
```

```java
import edu.columbia.apl2107.spaniel.procedures.Print;
import edu.columbia.apl2107.spaniel.procedures.Println;
import edu.columbia.apl2107.spaniel.procedures.REMatch;
import edu.columbia.apl2107.spaniel.procedures.Rest;
import edu.columbia.apl2107.spaniel.procedures.Subspans;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Root node class for the Custom AST.  Represents the entire program.
 *
 * @author alally
 */
public class Program extends AstNode
{
  private List _procedures = new ArrayList();
  private SymbolTable _symbolTable = new SymbolTable(); //stores procedure names

  public Program()
  {
    initPredefinedProcs();
  }

  public List getProcedures()
  {
    return Collections.unmodifiableList(_procedures);
  }

  public SymbolTable getSymbolTable()
  {
    return _symbolTable;
  }

  public void addProcedure(Procedure p)
  {
    String name = p.getName();
    assert name != null;
    if (_symbolTable.containsKey(name))
    {
      throw new StaticSemanticsException("Multiply-defined procedure " + name);
    }
    else
    {
      _symbolTable.put(name, p);
      _procedures.add(p);
    }
  }

  /**
   * Run the program.
   * @param doc the AnnotatedDocument on which this program should operate
   * @param args arguments
   */
  public void run(AnnotatedDocument doc, Value[] args)
  {
    //run main procedure
    Object mainProc = _symbolTable.get("main");
    assert mainProc instanceof Procedure;
    ((Procedure)mainProc).invoke(doc,args);
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    Iterator it = _procedures.iterator();
    while (it.hasNext())
    {
      buf.append(it.next()).append('\n');
    }
    return buf.toString();
  }
```

```
  private void initPredefinedProcs()
  {
    ArrayList oneParam = new ArrayList();
    oneParam.add("a");
    ArrayList twoParams = new ArrayList();
    twoParams.add("a");
    twoParams.add("b");

    _symbolTable.put("first", new ExternalProcedure("first",oneParam,First.class));
    _symbolTable.put("rest", new ExternalProcedure("rest",oneParam,Rest.class));
    _symbolTable.put("print", new ExternalProcedure("print",oneParam,Print.class));
    _symbolTable.put("println", new ExternalProcedure("print",oneParam,Println.class));
    _symbolTable.put("annotate", new
ExternalProcedure("annotate",twoParams,Annotate.class));
    _symbolTable.put("reMatch", new
ExternalProcedure("reMatch",twoParams,REMatch.class));
    _symbolTable.put("matching", new
ExternalProcedure("matching",twoParams,Matching.class));
    _symbolTable.put("subspans", new
ExternalProcedure("subspans",oneParam,Subspans.class));
    _symbolTable.put("instancesOf", new
ExternalProcedure("instancesOf",twoParams,InstancesOf.class));
  }
}
```

8.3.21 RelationalExpression.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.FloatValue;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for a relational (==, !=, &lt;, &lt;=, >, >=) expression.
 *
 * @author alally
 */
public class RelationalExpression extends BinOpExpression
{
  public RelationalExpression(int op, Expression lhs, Expression rhs)
  {
    super(op, lhs, rhs);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    Value v1 = getLhs().getValue(activationRecord).convertSequence();
    Value v2 = getRhs().getValue(activationRecord).convertSequence();

    switch(getOperator())
    {
      case SpanielVocabTokenTypes.LT:
        return BooleanValue.valueOf(lt(v1,v2));

      case SpanielVocabTokenTypes.LE:
        return BooleanValue.valueOf(lt(v1,v2) || eq(v1,v2));

      case SpanielVocabTokenTypes.GT:
        return BooleanValue.valueOf(gt(v1,v2));
```

```java
    case SpanielVocabTokenTypes.GE:
      return BooleanValue.valueOf(gt(v1,v2) || eq(v1,v2));

    case SpanielVocabTokenTypes.EQ:
      return BooleanValue.valueOf(eq(v1,v2));

    case SpanielVocabTokenTypes.NOTEQ:
      return BooleanValue.valueOf(!eq(v1,v2));

    default:
      assert false : "Unknown operator";
      return null;
  }
}

private boolean lt(Value v1, Value v2)
{
  if (v1 instanceof IntegerValue)
  {
    int int1 = ((IntegerValue)v1).getInt();
    if (v2 instanceof IntegerValue)
    {
      return  int1 < ((IntegerValue)v2).getInt();
    }
    else if (v2 instanceof FloatValue)
    {
      return  int1 < ((FloatValue)v2).getFloat();
    }
  }
  else if (v1 instanceof FloatValue)
  {
    float float1 = ((FloatValue)v1).getFloat();
    if (v2 instanceof IntegerValue)
    {
      return float1 < ((IntegerValue)v2).getInt();
    }
    else if (v2 instanceof FloatValue)
    {
      return float1 < ((FloatValue)v2).getFloat();
    }
  }
  else if (v1 instanceof SpanValue && v2 instanceof SpanValue)
  {
    return ((SpanValue)v1).compareTo(v2) < 0;
  }
  //fall-through if invalid types
  throw new SpanielRuntimeException("Invalid types to < operator: " +
      v1.getTypeName() + "," + v2.getTypeName());
}

private boolean eq(Value v1, Value v2)
{
  return v1.equals(v2);
}

private boolean gt(Value v1, Value v2)
{
  if (v1 instanceof IntegerValue)
  {
    int int1 = ((IntegerValue)v1).getInt();
    if (v2 instanceof IntegerValue)
    {
      return  int1 > ((IntegerValue)v2).getInt();
    }
    else if (v2 instanceof FloatValue)
    {
      return  int1 > ((FloatValue)v2).getFloat();
    }
  }
  else if (v1 instanceof FloatValue)
```

```
    {
      float float1 = ((FloatValue)v1).getFloat();
      if (v2 instanceof IntegerValue)
      {
        return float1 > ((IntegerValue)v2).getInt();
      }
      else if (v2 instanceof FloatValue)
      {
        return float1 > ((FloatValue)v2).getFloat();
      }
    }
    else if (v1 instanceof SpanValue && v2 instanceof SpanValue)
    {
      return ((SpanValue)v1).compareTo(v2) > 0;
    }
    //fall-through if invalid types
    throw new SpanielRuntimeException("Invalid types to > operator: " +
        v1.getTypeName() + "," + v2.getTypeName());
  }
}
```

8.3.22 SpanExpression.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;

/**
 * Custom AST node class for a Span Expression.
 *
 * @author alally
 */
public class SpanExpression extends Expression
{
  private Expression _begin;
  private Expression _end;

  public SpanExpression(Expression begin, Expression end)
  {
    _begin = begin;
    _end = end;
  }


  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    SpanValue span = new SpanValue();
    span.setBegin(_begin.getValue(activationRecord).convertSequence());
    span.setEnd(_end.getValue(activationRecord).convertSequence());
    return span;
  }


  public String toString()
  {
    return "(SPAN " + _begin + " " + _end + ")";
  }
}
```

8.3.23 Statement.java

```
/*
 * Created on Nov 10, 2004
```

```java
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.util.ActivationRecord;

/**
 * Custom AST node abstract class for a statement.  Subclassed by the various
 * types of statements.
 *
 * @author alally
 */
public abstract class Statement extends AstNode
{
  /**
   * Execute this statement.
   *
   * @param activationRecord the activation record for the currently executing procedure
   */
  public abstract void execute(ActivationRecord activationRecord);
}
```

8.3.24 SymbolTable.java

```java
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import java.util.HashMap;

/**
 * A Symbol table, which is a Map from String names to VarEntry or Procedure objects that
 * represent the Variable or Procedure that is bound to the name.
 * <p>
 * The entire {@link Program} has one symbol table, which holds all of the procedure
 * definitions.  Each {@link Procedure} then has a symbol table that holds the
 * parameter variables and local variables for that procedure.
 *
 * @author alally
 */
public class SymbolTable extends HashMap
{
  SymbolTable _parent;

  public SymbolTable()
  {
    this(null);
  }

  public SymbolTable(SymbolTable parent)
  {
    _parent = parent;
  }

  public SymbolTable getParent()
  {
    return _parent;
  }
}
```

8.3.25 UnaryExpression.java

```java
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.parser.SpanielVocabTokenTypes;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
```

```java
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.FloatValue;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node for an expression consisting of a unary operator (+,-,!) and
 * one operatnd.
 *
 * @author alally
 */
public class UnaryExpression extends Expression
{
  private int _operator;
  private Expression _operand;

  /**
   * Create a new UnaryExpression.
   *
   * @param op the operator.  The integer code used here is that generated by
   *   ANLTR in the class {@link SpanielVocabTokenTypes}.
   * @param operand the operand expression
   */
  public UnaryExpression(int operator, Expression operand)
  {
    _operator = operator;
    _operand = operand;
  }

  public String toString()
  {
    return "(" + getOperatorString() + " " + _operand + ")";
  }

  public String getOperatorString()
  {
    switch (_operator)
    {
      case SpanielVocabTokenTypes.UPLUS: return "+";
      case SpanielVocabTokenTypes.UMINUS: return "-";
      case SpanielVocabTokenTypes.NOT: return "!";
      default: return null;
    }
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Expression#eval(ActivationRecord)
   */
  public Object eval(ActivationRecord activationRecord)
  {
    Value v = getOperand().getValue(activationRecord).convertSequence();

    switch(getOperator())
    {
      case SpanielVocabTokenTypes.UPLUS:
        if (v instanceof IntegerValue || v instanceof FloatValue)
        {
          return v;
        }
        else
        {
          throw new SpanielRuntimeException("Invalid type to unary + operator: " +
v.getTypeName());
        }

      case SpanielVocabTokenTypes.UMINUS:
        if (v instanceof IntegerValue)
        {
          return new IntegerValue(-((IntegerValue)v).getInt());
        }
        else if (v instanceof FloatValue)
```

```
        {
          return new FloatValue(-((FloatValue)v).getFloat());
        }
        else
        {
          throw new SpanielRuntimeException("Invalid type to unary - operator: " +
v.getTypeName());
        }

      case SpanielVocabTokenTypes.NOT:
        v = v.coerceToBoolean();
        if (v instanceof BooleanValue)
        {
          return BooleanValue.valueOf(!((BooleanValue)v).getBoolean());
        }
        else
        {
          throw new SpanielRuntimeException("Invalid type to unary ! operator: " +
v.getTypeName());
        }

      default:
        assert false : "Unknown operator";
        return null;
    }
  }

  protected int getOperator()
  {
    return _operator;
  }

  protected Expression getOperand()
  {
    return _operand;
  }
}
```

8.3.26 VarEntry.java

```
/*
 * Created on Dec 9, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

/**
 * An entry in the symbol table for a variable.  Stores an index
 * into the current activation record.  To get the Variable to
 * which this VarEntry refers, call
 * <code>activationRecord.getLocalVariable(varEntry.getIndex())</code>.
 *
 * @author alally
 */
public class VarEntry
{
  private String _name;
  private int _index;

  /**
   * Create a new VarEntry
   * @param name name of the entry
   * @param index the index into the ActivationRecord of the
   *    Variable to which this VarEntry refers
   */
  public VarEntry(String name, int index)
  {
    _name = name;
    _index = index;
  }

  /**
```

```
   * Get the index into the ActivationRecord of the Variable
   * to which this VarEntry is bound.
   *
   * @return
   */
  public int getIndex()
  {
    return _index;
  }

  public String toString()
  {
    return (_name == null ? "var" : _name) + "@" + _index;
  }
}
```

8.3.27 WhileStatement.java

```
/*
 * Created on Nov 14, 2004
 */
package edu.columbia.apl2107.spaniel.ast;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.util.ActivationRecord;
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Custom AST node class for a forAll statement.
 *
 * @author alally
 */
public class WhileStatement extends Statement
{
  private Expression _condition;
  private Statement _body;

  public WhileStatement(Expression condition, Statement body)
  {
    _condition = condition;
    _body = body;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.ast.Statement#execute(ActivationRecord)
   */
  public void execute(ActivationRecord activationRecord)
  {
    while(!activationRecord.isBreakFlag())
    {
      Value conditionVal =
          _condition.getValue(activationRecord).coerceToBoolean();
      if (!(conditionVal instanceof  BooleanValue))
      {
        throw new SpanielRuntimeException(
            "Non-boolean value " + conditionVal + " in while statement condition");
      }
      if (!((BooleanValue)conditionVal).getBoolean())
      {
        break;
      }
      _body.execute(activationRecord);
    }
    activationRecord.setBreakFlag(false);
  }

  public String toString()
  {
    return "(while " + _condition.toString() + " " + _body.toString() + ")";
  }
```

```
}
```

8.4 Package edu.columbia.apl2107.spaniel.exception

8.4.1 InternalException.java

```java
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.exception;

/**
 * Exception thrown if an internal failure occurs in the Spaniel interpreter.
 *
 * @author alally
 */
public class InternalException extends RuntimeException
{
  /**
   *
   */
  public InternalException()
  {
    super();
  }

  /**
   * @param arg0
   */
  public InternalException(String arg0)
  {
    super(arg0);
  }

  /**
   * @param arg0
   */
  public InternalException(Throwable arg0)
  {
    super(arg0);
  }

  /**
   * @param arg0
   * @param arg1
   */
  public InternalException(String arg0, Throwable arg1)
  {
    super(arg0, arg1);
  }

}
```

8.4.2 SpanielRuntimeException.java

```java
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.exception;

/**
 * Exception thrown if a runtime error occurs in the Spaniel interpreter.
 * The most common type of runtime exception is a type mismatch when an operator or
 * procedure is evaluated.
 *
 * @author alally
 */
public class SpanielRuntimeException extends RuntimeException
{
```

```
  /**
   *
   */
  public SpanielRuntimeException()
  {
    super();
  }

  /**
   * @param arg0
   */
  public SpanielRuntimeException(String arg0)
  {
    super(arg0);
  }

  /**
   * @param arg0
   */
  public SpanielRuntimeException(Throwable arg0)
  {
    super(arg0);
  }

  /**
   * @param arg0
   * @param arg1
   */
  public SpanielRuntimeException(String arg0, Throwable arg1)
  {
    super(arg0, arg1);
  }

}
```

8.4.3 StaticSemanticsException.java

```
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.exception;

/**
 * Exception thrown when the Spaniel program is parsed, indicating
 * a violation of the static semantics constraints in the Spaniel language.
 * Examples are calls to undefined procedures or the existence of a break statement
 * outside of a loop.
 *
 * @author alally
 */
public class StaticSemanticsException extends RuntimeException
{
  /**
   *
   */
  public StaticSemanticsException()
  {
    super();
  }
  /**
   * @param message
   */
  public StaticSemanticsException(String message)
  {
    super(message);
  }
  /**
   * @param message
   * @param cause
   */
```

```
  public StaticSemanticsException(String message, Throwable cause)
  {
    super(message, cause);
  }
  /**
   * @param cause
   */
  public StaticSemanticsException(Throwable cause)
  {
    super(cause);
  }
}
```

8.5 Package edu.columbia.apl2107.spaniel.parser

All files in this package were generated by ANTLR and have been omitted from this report.

8.6 Package edu.columbia.apl2107.spaniel.procedures

8.6.1 Annotate.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that labels a Span and adds it to the AnnotatedDocument.
 *
 * @author alally
 */
public class Annotate implements JavaProcedure
{
  SpanValue _span;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 2; //should be enforced by static semantics
    arguments[0] = arguments[0].convertSequence();
    arguments[1] = arguments[1].convertSequence();
    if (arguments[0] instanceof SpanValue && arguments[1] instanceof StringValue)
    {
      _span = (SpanValue)arguments[0];
      //label the span
      _span.getField("type").setValue(arguments[1]);
      //check begin and end range
      int begin = _span.getBegin().getInt();
      int end = _span.getEnd().getInt();
      if (begin > end || begin < 0 || end > doc.getText().length())
      {
        throw new SpanielRuntimeException("Invalid range (" + begin + "," + end +
            ") in document of length " + doc.getText().length());
      }
      //post to annotated document
      doc.addAnnotation(_span);
```

```
    }
    else
    {
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + "," + arguments[1].getTypeName() + ") to annotate(Span,String).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    Value result = _span;
    _span = null;  //so we only return it once
    return result;
  }

}
```

8.6.2 First.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that gets the first element of a sequence.
 *
 * @author alally
 */
public class First implements JavaProcedure
{
  SequenceValue _seq;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 1; //should be enforced by static semantics
    if (arguments[0] instanceof SequenceValue)
    {
      _seq = (SequenceValue)arguments[0];
    }
    else
    {
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + ") to first(Sequence).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    if (_seq == null)
```

```
    {
      return null; //done
    }
    Value result = _seq.getHead();
    _seq = null;  //so we only return it once
    return result;
  }

}
```

8.6.3 InstancesOf.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import java.util.Iterator;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that gets retrieves subspans that have their <code>type</code>
 * field set to a specified value.
 *
 * @author alally
 */
public class InstancesOf implements JavaProcedure
{
  SpanValue _span;
  StringValue _type;
  Iterator _annotationIterator;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 2; //should be enforced by static semantics
    arguments[0] = arguments[0].convertSequence();
    arguments[1] = arguments[1].convertSequence();
    if (arguments[0] instanceof StringValue && arguments[1] instanceof SpanValue)
    {
      _type = (StringValue)arguments[0];
      _span = (SpanValue)arguments[1];
      _annotationIterator = doc.getAnnotationIterator();
    }
    else
    {
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + "," + arguments[1].getTypeName() + ") to instancesOf(String,Span).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    //TODO: what if annotations list modified between calls - this won't work
    while (_annotationIterator.hasNext())
    {
```

```
      SpanValue span = (SpanValue)_annotationIterator.next();
      if (span.isSubspan(_span) && span.getField("type").getValue().equals(_type))
        return span;
    }
    return null;
  }

}
```

8.6.4 Matching.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Built-in procedure that gets retrieves subspans that match a regular expression.
 *
 * @author alally
 */
public class Matching implements JavaProcedure
{
  Matcher _matcher;
  int _begin;
  int _end;
  int _current;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 2; //should be enforced by static semantics
    arguments[0] = arguments[0].convertSequence();
    arguments[1] = arguments[1].convertSequence();
    if (arguments[0] instanceof StringValue &&
        arguments[1] instanceof SpanValue)
    {
      String regex = ((StringValue)arguments[0]).getString();
      SpanValue span = (SpanValue)arguments[1];
      //check begin and end range
      _begin = span.getBegin().getInt();
      _end = span.getEnd().getInt();
      if (_begin > _end || _begin < 0 || _end > doc.getText().length())
      {
        throw new SpanielRuntimeException("Invalid range (" + _begin + "," + _end +
            ") in document of length " + doc.getText().length());
      }
      //compile regex
      Pattern pattern = Pattern.compile(regex);
      _matcher = pattern.matcher(doc.getText().substring(_begin,_end));
      _current = 0;
    }
    else
    {
```

```
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + "," + arguments[1].getTypeName() + ") to matching(String,Span).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    while (_matcher.find(_current))
    {
      SpanValue match = new SpanValue();
      match.setBegin(new IntegerValue(_begin + _matcher.start()));
      match.setEnd(new IntegerValue(_begin + _matcher.end()));
      //set _group field to the top-level group that matcher (0 if none)
      //for example if the regex is (foo)|(bar), then _group will be set to 1 if
      //foo matcher and 2 if bar matched.
      Variable groupField = match.createField("_group");
      groupField.setValue(new IntegerValue(0));
      for (int i = 1; i < _matcher.groupCount(); i++)
      {
        if (_matcher.group().equals(_matcher.group(i)))
        {
          groupField.setValue(new IntegerValue(i));
          break;
        }
      }
      _current = _matcher.end();
      return match;
    }
    //no more matches
    return null;
  }

}
```

8.6.5 Print.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;
import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that prints its argument to System.out.
 *
 * @author alally
 */
public class Print implements JavaProcedure
{
  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.var_value.Va
lue[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    System.out.print(arguments[0]);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
```

```
    // print does not return anything
    return null;
  }

}
```

8.6.6 Println.java

```
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;
import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that prints its argument, followed by a newline, to System.out.
 *
 * @author alally
 */
public class Println implements JavaProcedure
{
  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.var_value.Va
lue[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    System.out.println(arguments[0]);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    // print does not return anything
    return null;
  }

}
```

8.6.7 REMatch.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Built-in procedure that gets retrieves the first subspan that matches a regular
expression.
 *
 * @author alally
 */
public class REMatch implements JavaProcedure
```

```
{
  Matcher _matcher;
  int _begin;
  int _end;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 2; //should be enforced by static semantics
    arguments[0] = arguments[0].convertSequence();
    arguments[1] = arguments[1].convertSequence();
    if (arguments[0] instanceof StringValue &&
        arguments[1] instanceof SpanValue)
    {
      String regex = ((StringValue)arguments[0]).getString();
      SpanValue span = (SpanValue)arguments[1];
      //check begin and end range
      _begin = span.getBegin().getInt();
      _end = span.getEnd().getInt();
      if (_begin > _end || _begin < 0 || _end > doc.getText().length())
      {
        throw new SpanielRuntimeException("Invalid range (" + _begin + "," + _end +
            ") in document of length " + doc.getText().length());
      }
      //compile regex
      Pattern pattern = Pattern.compile(regex);
      _matcher = pattern.matcher(doc.getText().substring(_begin,_end));
    }
    else
    {
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + "," + arguments[1].getTypeName() + ") to reMatch(String,Span).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    if (_matcher == null)
    {
      return null;
    }
    if (_matcher.find())
    {
      SpanValue match = new SpanValue();
      match.setBegin(new IntegerValue(_begin + _matcher.start()));
      match.setEnd(new IntegerValue(_begin + _matcher.end()));
      //set _group field to the top-level group that matcher (0 if none)
      //for example if the regex is (foo)|(bar), then _group will be set to 1 if
      //foo matcher and 2 if bar matched.
      Variable groupField = match.createField("_group");
      groupField.setValue(new IntegerValue(0));
      for (int i = 1; i < _matcher.groupCount(); i++)
      {
        if (_matcher.group().equals(_matcher.group(i)))
        {
          groupField.setValue(new IntegerValue(i));
          break;
        }
      }
      _matcher = null; //so we only return the first match
      return match;
    }
```

```
    else //no more matches
    {
      return null;
    }
  }

}
```

8.6.8 Rest.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that gets all but the first element of a sequence.
 *
 * @author alally
 */
public class Rest implements JavaProcedure
{
  SequenceValue _seq;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 1; //should be enforced by static semantics
    if (arguments[0] instanceof SequenceValue)
    {
      _seq = (SequenceValue)arguments[0];
      _seq = _seq.getTail();
    }
    else
    {
      throw new SpanielRuntimeException("Invalid arguments (" +
arguments[0].getTypeName()
          + ") to first(Sequence).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    if (_seq == null)
    {
      return null; //done
    }
    Value result = _seq.getHead();
    _seq = _seq.getTail();
    return result;
  }

}
```

8.6.9 Subspans.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.procedures;

import java.util.Iterator;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * Built-in procedure that gets all subspans of the specified span that are
 * have been posted to the AnnotatedDocument.
 *
 * @author alally
 */
public class Subspans implements JavaProcedure
{
  SpanValue _span;
  Iterator _annotationIterator;

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    assert arguments.length == 1; //should be enforced by static semantics
    arguments[0] = arguments[0].convertSequence();
    if (arguments[0] instanceof SpanValue)
    {
      _span = (SpanValue)arguments[0];
      _annotationIterator = doc.getAnnotationIterator();
    }
    else
    {
      throw new SpanielRuntimeException("Invalid argument (" + arguments[0].getTypeName()
          + ") to subspans(Span).");
    }

  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    //TODO: this works for now because we're sure that the annotations list will not
    //be modified between calls.  But this should be revisited if procedure invocation
    //is reimplemented to a call-on-demand style.  If the annotations list is modified
    //between calls, this will fail.
    while (_annotationIterator.hasNext())
    {
      SpanValue span = (SpanValue)_annotationIterator.next();
      if (span.isSubspan(_span))
        return span;
    }
    return null;
  }

}
```

8.7 Package edu.columbia.apl2107.spaniel.test

8.7.1 JUnitAsser.java

```
/*
 * Created on Dec 14, 2004
 */
package edu.columbia.apl2107.spaniel.test;

import junit.framework.Assert;
import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.BooleanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * A Spaniel JavaProcedure that invokes the JUnit Assert.assertTrue method.
 * This allows a Spaniel program to essentially become a JUnit test case.
 *
 * @author alally
 */
public class JUnitAssert implements JavaProcedure
{

  /**
   *
   */
  public JUnitAssert()
  {
    super();
  }

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    if (arguments.length != 1)
    {
      throw new SpanielRuntimeException("Procedure JUnitAssert requires 1 argument.");
    }
    arguments[0] = arguments[0].coerceToBoolean();
    if (arguments[0] instanceof BooleanValue)
    {
      Assert.assertTrue(((BooleanValue)arguments[0]).getBoolean());
    }
    else
    {
      throw new SpanielRuntimeException("Invalid argument (" + arguments[0].getTypeName()
          + ") to JUnitAssert(Boolean).");
    }
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    return null;
  }

}
```

8.7.2 JUnitAssertEquals.java

```
/*
 * Created on Dec 14, 2004
```

```
 */
package edu.columbia.apl2107.spaniel.test;

import junit.framework.Assert;
import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.JavaProcedure;
import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;
import edu.columbia.apl2107.spaniel.var_value.Value;

/**
 * A Spaniel JavaProcedure that invokes the JUnit Assert.assertEquals method.
 * This allows a Spaniel program to essentially become a JUnit test case.
 *
 * @author alally
 */
public class JUnitAssertEquals implements JavaProcedure
{

  /**
   *
   */
  public JUnitAssertEquals()
  {
    super();
  }

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.JavaProcedure#init(edu.columbia.apl2107.spaniel.AnnotatedDoc
ument, edu.columbia.apl2107.spaniel.var_value.Value[])
   */
  public void init(AnnotatedDocument doc, Value[] arguments)
  {
    if (arguments.length != 2)
    {
      throw new SpanielRuntimeException("Procedure JUnitAssertEquals requires 2
arguments.");
    }
    arguments[0] = arguments[0].convertSequence();
    arguments[1] = arguments[1].convertSequence();
    Assert.assertEquals(arguments[0], arguments[1]);
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.JavaProcedure#next()
   */
  public Value next()
  {
    return null;
  }

}
```

8.7.3 JUnitTestCase.java

```
/*
 * Created on Dec 14, 2004
 */
package edu.columbia.apl2107.spaniel.test;

import java.util.Iterator;

import junit.framework.TestCase;
import edu.columbia.apl2107.spaniel.Interpreter;
import edu.columbia.apl2107.spaniel.util.AnnotatedDocumentImpl;
import edu.columbia.apl2107.spaniel.var_value.IntegerValue;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.StringValue;

/**
 * A JUnit test case that runs several Spaniel programs.  These programs are written
```

```
 * so that they call (via javacall) the JUnit assert methods.
 *
 * @author alally
 */
public class JUnitTestCase extends TestCase
{
  public void testMath() throws Exception
  {
    Interpreter.run("test/junit/math.spl", new AnnotatedDocumentImpl("Hello World!"));
  }

  public void testSpans() throws Exception
  {
    Interpreter.run("test/junit/spans.spl", new AnnotatedDocumentImpl("Hello World!"));
  }

  public void testSequences() throws Exception
  {
    Interpreter.run("test/junit/sequences.spl", new AnnotatedDocumentImpl("Hello
World!"));
  }

  public void testControlFlow() throws Exception
  {
    Interpreter.run("test/junit/controlFlow.spl", new AnnotatedDocumentImpl("Hello
World!"));
  }

  public void testRecursion() throws Exception
  {
    Interpreter.run("test/junit/recursion.spl", new AnnotatedDocumentImpl("Hello
World!"));
  }

  public void testAnnotation() throws Exception
  {
    AnnotatedDocumentImpl doc = new AnnotatedDocumentImpl("Hello World!");
    Interpreter.run("test/junit/annotation.spl", doc);

    //also test that AnnotatedDocument contains the expected results
    assertEquals("Hello World!", doc.getText());
    Iterator it = doc.getAnnotationIterator();
    SpanValue a0 = (SpanValue)it.next();
    assertEquals(a0.getBegin(), new IntegerValue(0));
    assertEquals(a0.getEnd(), new IntegerValue(5));
    assertEquals(a0.getField("type").getValue(), new StringValue("Greeting"));
    SpanValue a1 = (SpanValue)it.next();
    assertEquals(a1.getBegin(), new IntegerValue(0));
    assertEquals(a1.getEnd(), new IntegerValue(1));
    assertEquals(a1.getField("type").getValue(), new StringValue("Capital Letter"));
    SpanValue a2 = (SpanValue)it.next();
    assertEquals(a2.getBegin(), new IntegerValue(6));
    assertEquals(a2.getEnd(), new IntegerValue(11));
    assertEquals(a2.getField("type").getValue(), new StringValue("Place"));
    SpanValue a3 = (SpanValue)it.next();
    assertEquals(a3.getBegin(), new IntegerValue(6));
    assertEquals(a3.getEnd(), new IntegerValue(7));
    assertEquals(a3.getField("type").getValue(), new StringValue("Capital Letter"));
    assertFalse(it.hasNext());
  }
}
```

8.7.4 TestParser.java

```
/*
 * Created on Oct 3, 2004
 */
package edu.columbia.apl2107.spaniel.test;

import java.io.File;
import java.io.FileInputStream;
```

```java
import java.io.FileWriter;
import java.io.InputStream;
import java.io.Writer;

import antlr.BaseAST;
import edu.columbia.apl2107.spaniel.parser.SpanielLexer;
import edu.columbia.apl2107.spaniel.parser.SpanielParser;

/**
 * Parses all files in the test/input directory and produces AST .xml files in the
 * test/ast directory.
 *
 * @author alally
 */
public class TestParser
{

  public static void main(String[] args)
  {
    try
    {
      File inputDir = new File("test/input");
      File[] inputFiles = inputDir.listFiles();
      for (int i = 0; i < inputFiles.length; i++)
      {
        System.out.println("Parsing " + inputFiles[i].getName());
        InputStream input = new FileInputStream(inputFiles[i]);
        SpanielLexer lexer = new SpanielLexer(input);
        SpanielParser parser = new SpanielParser(lexer);
        parser.program();
        File outputFile = new File("test/ast/" + inputFiles[i].getName()
            + ".xml");
        Writer writer = new FileWriter(outputFile);
        ((BaseAST) parser.getAST()).xmlSerialize(writer);
        writer.close();
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

8.7.5 TestTreeRewriter.java

```java
/*
 * Created on Oct 3, 2004
 */
package edu.columbia.apl2107.spaniel.test;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

import antlr.BaseAST;
import edu.columbia.apl2107.spaniel.ast.Program;
import edu.columbia.apl2107.spaniel.parser.SpanielLexer;
import edu.columbia.apl2107.spaniel.parser.SpanielParser;
import edu.columbia.apl2107.spaniel.parser.SpanielTreeRewriter;

/**
 * Parses all files in the test/input directory and invokes the tree rewriter (walker)
 * to produce the Custom Spaniel AST.  The String representation of this (an informal
 * LISP-like syntax) is printed to System.out.
 *
 * @author alally
 */
public class TestTreeRewriter
{

  public static void main(String[] args)
```

```
  {
    try
    {
      File inputDir = new File("test/input");
      File[] inputFiles = inputDir.listFiles();
      for (int i = 0; i < inputFiles.length; i++)
      {
        System.out.println("Parsing " + inputFiles[i].getName());
        InputStream input = new FileInputStream(inputFiles[i]);
        SpanielLexer lexer = new SpanielLexer(input);
        SpanielParser parser = new SpanielParser(lexer);
        parser.program();
        BaseAST ast = (BaseAST) parser.getAST();
        SpanielTreeRewriter walker = new SpanielTreeRewriter();
        Program prog = walker.program(ast);
        System.out.println(prog.toString());
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

8.7.6 ViewAST.java

```
/*
 * Created on Oct 3, 2004
 */
package edu.columbia.apl2107.spaniel.test;

import java.io.FileInputStream;
import java.io.InputStream;

import antlr.CommonAST;
import antlr.debug.misc.ASTFrame;
import edu.columbia.apl2107.spaniel.parser.SpanielLexer;
import edu.columbia.apl2107.spaniel.parser.SpanielParser;

/**
 * Test program for debugging the ANTLR-generated CommonAST.  Takes one argument, the
path
 * to a Spaniel program file.  Parses this and builds the AST, then displays it in a
 * GUI tree viewer.
 *
 * @author alally
 */
public class ViewAST
{

  public static void main(String[] args)
  {
    try
    {
      InputStream input = new FileInputStream(args[0]);
      SpanielLexer lexer = new SpanielLexer(input);
      SpanielParser parser = new SpanielParser(lexer);
      parser.program();
      // Get the AST from the parser
      CommonAST parseTree = (CommonAST) parser.getAST();

      // Print the AST in a human-readable format
      System.out.println(parseTree.toStringList());

      // Open a window in which the AST is displayed graphically
      ASTFrame frame = new ASTFrame("Spaniel AST", parseTree);
      frame.setVisible(true);

    } catch (Exception e)
    {
      e.printStackTrace();
```

```
    }
  }
}
```

8.8 Package edu.columbia.apl2107.spaniel.util

8.8.1 ActivationRecord.java

```
/*
 * Created on Dec 9, 2004
 */
package edu.columbia.apl2107.spaniel.util;

import java.util.ArrayList;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.ast.Procedure;
import edu.columbia.apl2107.spaniel.var_value.SequenceValue;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
 * Holds all local variables of an executing procedure.  An ActivationRecord is
 * created whenever a procedure is invoked.  Note that Spaniel doesn't declare its
 * own stack to hold activation records; they are stored on the Java call stack as
 * arguments to {@link Expression#eval(ActivationRecord)}.
 * <p>
 * The ActivationRecord also conains:
 * <ul>
 *   <li>The sequence that is the implicit return value of all procedures, and to which
 *       values are appended by the <code>emit</code> statement.</li>
 *   <li>A flag indicating whether we are currently breaking out of a loop due to an
 *       execution of the <code>break</code> statement.</li>
 *   <li>A reference to the AnnotatedDocument on which the procedure operates.
 *       This is needed by many of the built-in functions.</li>
 * </ul>
 *
 * @author alally
 */
public class ActivationRecord
{
  private ArrayList _variables = new ArrayList();
  private SequenceValue _returnValue = new SequenceValue();
  private boolean _breakFlag;
  private AnnotatedDocument _annotatedDocument;


  /**
   * Create an ActivationRecord for an invocation of the given procedure
   * @param proc the Procedure invoked
   * @param doc the AnnotatedDocument on which the procedure invocation should operate
   * @param arguments arguments passed to the procedure
   */
  public ActivationRecord(Procedure proc, AnnotatedDocument doc, Value[] arguments)
  {
    //create variables for VarEntries in the symbol table
    int numVars = proc.getSymbolTable().size(); //proc symbol table only contains
VarEntries
    assert numVars >= arguments.length;
    for (int i = 0; i < numVars; i++)
    {
      Variable var = new Variable();
      //the first arguments.length variables get initialized with the procedure arguments
      if (i < arguments.length)
      {
        var.setValue(arguments[i]);
      }
      _variables.add(i, var);
    }
    _annotatedDocument = doc;
```

```
  }

  public Variable getLocalVariable(int index)
  {
    return (Variable)_variables.get(index);
  }

  public SequenceValue getReturnValue()
  {
    return _returnValue;
  }

  public void setBreakFlag(boolean aFlag)
  {
    _breakFlag = aFlag;
  }

  public boolean isBreakFlag()
  {
    return _breakFlag;
  }

  public AnnotatedDocument getAnnotatedDocument()
  {
    return _annotatedDocument;
  }
}
```

8.8.2 AnnotatedDocumentImpl.java

```
/*
 * Created on Dec 13, 2004
 */
package edu.columbia.apl2107.spaniel.util;

import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;

/**
 * Implementation of {@link AnnotatedDocument}.
 *
 * @author alally
 */
public class AnnotatedDocumentImpl implements AnnotatedDocument
{
  private String _text;
  private List _annotations = new LinkedList();

  /**
   *
   */
  public AnnotatedDocumentImpl(String text)
  {
    super();
    _text = text;
  }

  /* (non-Javadoc)
   * @see edu.columbia.apl2107.spaniel.AnnotatedDocument#getText()
   */
  public String getText()
  {
    return _text;
  }
```

```java
  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.AnnotatedDocument#addAnnotation(edu.columbia.apl2107.spaniel
.var_value.SpanValue)
   */
  public void addAnnotation(SpanValue span)
  {
    _annotations.add(span);
    Collections.sort(_annotations);
  }


  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.AnnotatedDocument#getAnnotationIterator(edu.columbia.apl2107
.spaniel.var_value.SpanValue)
   */
  public ListIterator getAnnotationIterator()
  {
    return _annotations.listIterator();
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    buf.append(_text).append('\n');
    Iterator it = _annotations.iterator();
    while (it.hasNext())
    {
      buf.append(it.next()).append('\n');
    }
    return buf.toString();
  }
}
```

8.8.3 InlineXmlOutput.java

```java
/*
 * Created on Dec 16, 2004
 */
package edu.columbia.apl2107.spaniel.util;

import java.util.ListIterator;

import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.AttributesImpl;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;

/**
 * A utility class for generating inline XML output from an AnnotatedDocument.
 * Writes to a ContentHandler which can then be passed (for example) to the Xerces
 * XML Serializer to generate nicely formatted XML.
 *
 * @author alally
 */
public class InlineXmlOutput
{
  ContentHandler _contentHandler;

  /**
   *
   */
  public InlineXmlOutput(ContentHandler contentHandler)
  {
    super();
    _contentHandler = contentHandler;
  }
```

```
  public void generateXml(AnnotatedDocument doc) throws SAXException
  {
    _contentHandler.startDocument();
    _contentHandler.startElement("","Document","Document", new AttributesImpl());
    int pos = 0;
    ListIterator it = doc.getAnnotationIterator();
    generateXml1(doc.getText().toCharArray(), pos, doc.getText().length(), it);

    _contentHandler.endElement("","Document","Document");
    _contentHandler.endDocument();
  }

  private void generateXml1(char[] text, int start, int end, ListIterator annotations)
    throws SAXException
  {
    if (!annotations.hasNext()) //done
    {
      _contentHandler.characters(text, start, end - start);
      return;
    }

    SpanValue annotation = (SpanValue)annotations.next();
    int annotStart = annotation.getBegin().getInt();
    int annotEnd = annotation.getEnd().getInt();
    //is next annotation within the span of the current annotation?
    if (annotStart >= start && annotEnd <= end)
    {
      //yes - add characters up to start of annotation, then produce XML for the
annotation
      _contentHandler.characters(text, start, annotStart - start);
      String annotType = annotation.getField("type").getValue().toString();
      _contentHandler.startElement("",annotType, annotType, new AttributesImpl()); //TODO
- attributes
      generateXml1(text, annotation.getBegin().getInt(), annotation.getEnd().getInt(),
annotations);
      _contentHandler.endElement("",annotType, annotType);
      //now call self recrusively to handle next annotation
      generateXml1(text, annotEnd, end, annotations);
    }
    //is next annotation completely after current annotation?
    else if (annotStart >= end)
    {
      //yes - so finish characters and then push back iterator and return, so current
annotation will finish
      //being written before next is handled
      _contentHandler.characters(text, start, end - start);
      annotations.previous();
    }
  }
}
```

8.8.4 UimaXmlOutput.java

```
/*
 * Created on Dec 16, 2004
 */
package edu.columbia.apl2107.spaniel.util;

import java.util.Iterator;
import java.util.Map;

import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.AttributesImpl;

import edu.columbia.apl2107.spaniel.AnnotatedDocument;
import edu.columbia.apl2107.spaniel.var_value.SpanValue;
import edu.columbia.apl2107.spaniel.var_value.Value;
import edu.columbia.apl2107.spaniel.var_value.Variable;

/**
```

```java
 * A utility class for generating UIMA-compliant XML output from an AnnotatedDocument.
 * For information on UIMA see <a href="http://alphaworks.ibm.com/tech/uima">
 * http://alphaworks.ibm.com/tech/uima</a>.
 * Writes to a ContentHandler which can then be passed (for example) to the Xerces
 * XML Serializer to generate nicely formatted XML.
 *
 * @author alally
 */
public class UimaXmlOutput
{
  ContentHandler _contentHandler;

  /**
   *
   */
  public UimaXmlOutput(ContentHandler contentHandler)
  {
    super();
    _contentHandler = contentHandler;
  }

  public void generateXml(AnnotatedDocument doc) throws SAXException
  {
    AttributesImpl noAttrs = new AttributesImpl();
    _contentHandler.startDocument();
    _contentHandler.startElement("","CAS","CAS", noAttrs);
    //encode document
    _contentHandler.startElement("","uima.tcas.Document","uima.tcas.Document", noAttrs);
    _contentHandler.characters(doc.getText().toCharArray(), 0, doc.getText().length());
    _contentHandler.endElement("","uima.tcas.Document","uima.tcas.Document");

    //encode annotations
    Iterator it = doc.getAnnotationIterator();
    while (it.hasNext())
    {
      SpanValue span = (SpanValue)it.next();
      String label = span.getField("type").getValue().toString();
      AttributesImpl attrs = new AttributesImpl();
      attrs.addAttribute("","begin","begin","CDATA",span.getBegin().toString());
      attrs.addAttribute("","end","end","CDATA",span.getEnd().toString());
      attrs.addAttribute("","_indexed","_indexed","CDATA","1");
      //add other attributes
      Iterator entryIterator = span.getFieldMap().entrySet().iterator();
      while (entryIterator.hasNext())
      {
        Map.Entry entry = (Map.Entry)entryIterator.next();
        String name = (String)entry.getKey();
        Variable fieldVar = (Variable)entry.getValue();
        if (!name.equals("begin") && !name.equals("end"))
        {
          Value val = fieldVar.getValue();
          String valStr;
          if (val instanceof SpanValue)
          {
            SpanValue s = (SpanValue)val;
            valStr = doc.getText().substring(s.getBegin().getInt(), s.getEnd().getInt());
            //TODO: correctly handle references from one span object to another
          }
          else
          {
            valStr = val.toString();
          }
          attrs.addAttribute("",name,name,"CDATA",valStr);
        }
      }
      _contentHandler.startElement("",label,label,attrs);
      _contentHandler.endElement("",label,label);
    }

    _contentHandler.endElement("","CAS","CAS");
    _contentHandler.endDocument();
```

```
  }
}
```

8.9 Package edu.columbia.apl2107.spaniel.var_value

8.9.1 BooleanValue.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * A value of type Boolean.  There are only two instances of this class.  Use
 * {@link #valueOf(boolean)} to get an instance.
 *
 * @author alally
 */
public class BooleanValue extends Value
{
   boolean _val;

  /**
   * @param type
   */
  private BooleanValue(boolean val)
  {
    super(Type.BOOLEAN);
    _val = val;
  }

  public static BooleanValue valueOf(boolean val)
  {
    return val ? TRUE : FALSE;
  }

  public boolean getBoolean()
  {
    return _val;
  }

  public String toString()
  {
    return Boolean.toString(_val);
  }

  public boolean equals(Object o)
  {
    return (o instanceof BooleanValue) && (((BooleanValue)o).getBoolean() ==
getBoolean());
  }

  public int hashCode()
  {
    return _val ? 1 : 0;
  }

  public static final BooleanValue TRUE = new BooleanValue(true);
  public static final BooleanValue FALSE = new BooleanValue(false);
}
```

8.9.2 FloatValue.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
```

```java
 * A value of type Float.
 *
 * @author alally
 */
public class FloatValue extends Value
{
  float _val;

  /**
   * @param type
   */
  public FloatValue(float val)
  {
    super(Type.FLOAT);
    _val = val;
  }

  public float getFloat()
  {
    return _val;
  }

  public String toString()
  {
    return Float.toString(_val);
  }

  public boolean equals(Object o)
  {
    return (o instanceof FloatValue) && (((FloatValue)o).getFloat() == getFloat());
  }

  public int hashCode()
  {
    return Float.floatToIntBits(_val);
  }
}
```

8.9.3 IntegerValue.java

```java
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * A value of type Integer.
 *
 * @author alally
 */
public class IntegerValue extends Value
{
  int _val;

  /**
   * @param type
   */
  public IntegerValue(int val)
  {
    super(Type.INTEGER);
    _val = val;
  }

  public int getInt()
  {
    return _val;
  }

  public String toString()
  {
    return Integer.toString(_val);
```

```
  }

  public boolean equals(Object o)
  {
    return (o instanceof IntegerValue) && (((IntegerValue)o).getInt() == getInt());
  }

  public int hashCode()
  {
    return _val;
  }
}
```

8.9.4 NullValue.java

```
/*
 * Created on Dec 9, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * The null value.  This is a singleton, since all null values are equivalent.
 * Use {@link #instance()} to get the singleton instance.
 *
 * @author alally
 */
public class NullValue extends Value
{
  static final NullValue _instance = new NullValue();

  /**
   * @param type
   */
  private NullValue()
  {
    super(null);
  }

  public static NullValue instance()
  {
    return _instance;
  }

  public String toString()
  {
    return "null";
  }

  public int hashCode()
  {
    return 0;
  }
}
```

8.9.5 SequenceValue.java

```
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

import java.util.LinkedList;

/**
 * A value of type Sequence.
 *
 * @author alally
 */
public class SequenceValue extends Value
{
```

```java
LinkedList _sequence;

public SequenceValue()
{
  this(new LinkedList());
}

public SequenceValue(LinkedList sequence)
{
  super(Type.SEQUENCE);
  _sequence = sequence;
}

public boolean isEmpty()
{
  return _sequence.isEmpty();
}

public Value getHead()
{
  return _sequence.isEmpty() ? null : (Value)_sequence.getFirst();
}

public SequenceValue getTail()
{
  LinkedList tail = new LinkedList(_sequence);
  if (!tail.isEmpty())
    tail.removeFirst();
  return new SequenceValue(tail);
  //TODO: is there a way to do this without copy?
}

public void append(Value val)
{
  _sequence.add(val);
}

/* (non-Javadoc)
 * @see edu.columbia.apl2107.spaniel.var_value.Value#convertSequence()
 */
public Value convertSequence()
{
  if (_sequence.size() == 1)
    return (Value)_sequence.getFirst();
  else
    return this;
}

/* (non-Javadoc)
 * @see edu.columbia.apl2107.spaniel.var_value.Value#coerceToBoolean()
 */
public Value coerceToBoolean()
{
  if (_sequence.size() == 1 && _sequence.get(0) instanceof BooleanValue)
  {
    return (BooleanValue) _sequence.get(0);
  }
  else
  {
    return isEmpty() ? BooleanValue.FALSE : BooleanValue.TRUE;
  }
}

public String toString()
{
  return _sequence.toString();
}

public boolean equals(Object o)
{
```

```
    return (o instanceof SequenceValue) &&
(_sequence.equals(((SequenceValue)o)._sequence));
  }

  public int hashCode()
  {
    return _sequence.hashCode();
  }
}
```

8.9.6 SpanValue.java

```
/*
 * Created on Nov 21, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;

/**
 * A value of type Span.
 *
 * @author alally
 */
public class SpanValue extends Value implements Comparable
{
  /** Map from field name to Variable that represents that field */
  TreeMap _fields = new TreeMap();

  /** "begin" field has spcial status. */
  TypedVariable _beginField;

  /** "end" field has spcial status. */
  TypedVariable _endField;

  public SpanValue()
  {
    super(Type.SPAN);
    _beginField = new TypedVariable(Type.INTEGER);
    _fields.put("begin", _beginField);
    _endField = new TypedVariable(Type.INTEGER);
    _fields.put("end", _endField);
    _fields.put("type", new TypedVariable(Type.STRING));
  }

  public SpanValue(IntegerValue begin, IntegerValue end)
  {
    this();
    _beginField.setValue(begin);
    _endField.setValue(end);
  }

  /**
   * @return
   */
  public IntegerValue getBegin()
  {
    assert _beginField != null;
    return (IntegerValue)_beginField.getValue();
  }

  /**
   * Set the begin field of this span.
   * @param val value to set.  Must be an IntegerValue or a SpanielRuntimeException will
be thrown
   */
  public void setBegin(Value val)
```

```
  {
    assert _beginField != null;
    _beginField.setValue(val);
  }

  /**
   * @return
   */
  public IntegerValue getEnd()
  {
    assert _endField != null;
    return (IntegerValue)_endField.getValue();
  }

  /**
   * Set the end field of this span.
   * @param val value to set.  Must be an IntegerValue or a SpanielRuntimeException will
be thrown
   */
  public void setEnd(Value val)
  {
    assert _endField != null;
    _endField.setValue(val);
  }

  /**
   * Gets the Variable that is referenced by the named field.
   * @param name name of field
   * @return Variable object that is referenced by the named field
   */
  public Variable getField(String name)
  {
    Variable var = (Variable)_fields.get(name);
    return var;
  }

  /**
   * Create a new field with the specified name
   *
   * @param name name of the new field
   * @return the Variable for the new field
   */
  public Variable createField(String name)
  {
    assert getField(name) == null;
    Variable var = new Variable();
    _fields.put(name, var);
    return var;
  }

  /**
   * Gets a Map from field name to Variable object that stores the value of that field,
   * for each field defined in this span.
   *
   * @return a Map from field name to Variable
   */
  public Map getFieldMap()
  {
    return _fields;
  }

  public String toString()
  {
    StringBuffer buf = new StringBuffer();
    buf.append("[").append("begin=").append(getBegin());
    buf.append(",end=").append(getEnd());
    Iterator entryIterator = _fields.entrySet().iterator();
    while (entryIterator.hasNext())
    {
      Map.Entry entry = (Map.Entry)entryIterator.next();
      String name = (String)entry.getKey();
```

```java
    Variable fieldVar = (Variable)entry.getValue();
    if (!name.equals("begin") && !name.equals("end"))
    {
      buf.append(',').append(name).append('=').append(fieldVar.getValue());
    }
  }
  buf.append("]");
  return buf.toString();
}

public boolean equals(Object o)
{
  //as defined in LRM, equal means having begin and end values that are equal
  if (o instanceof SpanValue)
  {
    SpanValue s = (SpanValue)o;
    return this.getBegin().equals(s.getBegin()) && this.getEnd().equals(s.getEnd());
  }
  else
  {
    return false;
  }
}

public int hashCode()
{
  return getBegin().hashCode() ^ getEnd().hashCode();
}

/**
 * Determines if this span is a subspan of another span.
 * Span s is a subspan of Span t if s.begin >= t.begin and
 * s.end &lt;= t.end.
 *
 * @param superSpan the Span that must be a superspan of this Span in order
 *   for this method to return true
 * @return if this span is a subspan of <code>superSpan</code>.
 */
public boolean isSubspan(SpanValue superSpan)
{
  int s_beg = this.getBegin().getInt();
  int s_end = this.getEnd().getInt();
  int t_beg = superSpan.getBegin().getInt();
  int t_end = superSpan.getEnd().getInt();
  return s_beg >= t_beg && s_end <= t_end;
}

/* (non-Javadoc)
 * @see java.lang.Comparable#compareTo(java.lang.Object)
 */
public int compareTo(Object arg0)
{
  if (arg0 instanceof SpanValue)
  {
    SpanValue s1 = this;
    SpanValue s2 = (SpanValue)arg0;
    int b1 = s1.getBegin().getInt();
    int e1 = s1.getEnd().getInt();
    int b2 = s2.getBegin().getInt();
    int e2 = s2.getEnd().getInt();
    if (b1 == b2 && e1 == e2)
    {
      return 0;
    }
    else if (b1 < b2 || (b1 == b2 && e1 > e2))
    {
      return -1;
    }
    else
    {
      return 1;
```

```
      }
    }
    else
    {
      throw new SpanielRuntimeException("SpanValue compared to non-SpanValue");
    }
  }
}
```

8.9.7 StringValue.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * A value of type String.
 *
 * @author alally
 */
public class StringValue extends Value
{
   String _val;

  /**
   * @param type
   */
  public StringValue(String val)
  {
    super(Type.STRING);
    _val = val;
  }

  public String getString()
  {
    return _val;
  }

  public String toString()
  {
    return getString();
  }


  public boolean equals(Object o)
  {
    return (o instanceof StringValue) &&
(((StringValue)o).getString().equals(getString()));
  }

  public int hashCode()
  {
    return _val.hashCode();
  }

}
```

8.9.8 Type.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * Represents a type.  This is a type-safe enum.  You cannot create new instances of this
 * class; instead use the static constants INTEGER, FLOAT, etc.
 *
 * @author alally
```

```
 */
public class Type
{
  private int _code;

  private Type(int code)
  {
    _code = code;
  }

  public int getCode()
  {
    return _code;
  }

  public static final int _INTEGER = 0;
  public static final int _FLOAT = 1;
  public static final int _BOOLEAN = 2;
  public static final int _STRING = 3;
  public static final int _SPAN = 4;
  public static final int _SEQUENCE = 5;

  public static final Type INTEGER = new Type(_INTEGER);
  public static final Type FLOAT = new Type(_FLOAT);
  public static final Type BOOLEAN = new Type(_BOOLEAN);
  public static final Type STRING = new Type(_STRING);
  public static final Type SPAN = new Type(_SPAN);
  public static final Type SEQUENCE = new Type(_SEQUENCE);

  public String getName()
  {
    switch(_code)
    {
      case _INTEGER: return "integer";
      case _FLOAT: return "float";
      case _BOOLEAN: return "boolean";
      case _STRING: return "string";
      case _SPAN: return "span";
      case _SEQUENCE: return "sequence";
      default:
         assert false;
         return null;
    }
  }
}
```

8.9.9 TypedVariable.java

```
/*
 * Created on Dec 10, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

import edu.columbia.apl2107.spaniel.exception.SpanielRuntimeException;

/**
 * A Variable that imposes a type restriction on its value.  This is used
 * only in a few special cases in the Spaniel language.
 *
 * @author alally
 */
public class TypedVariable extends Variable
{
  Type _type;

  /**
   *
   */
  public TypedVariable(Type type)
  {
    this(null,type);
```

```
  }

  /**
   * @param name
   */
  public TypedVariable(String name, Type type)
  {
    super(name);
    _type = type;
  }

  /* (non-Javadoc)
   * @see
edu.columbia.apl2107.spaniel.var_value.Variable#setValue(edu.columbia.apl2107.spaniel.var
_value.Value)
   */
  public void setValue(Value value)
  {
    if (value.getType() == _type || value instanceof NullValue)
    {
      super.setValue(value);
    }
    else
    {
      throw new SpanielRuntimeException("Value of type " + value.getTypeName() +
          "cannot be assigned to variable " + getName() != null ? getName() : "");
    }
  }
}
```

8.9.10 Value.java

```
/*
 * Created on Nov 13, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * Abstract base class for all values.
 *
 * @author alally
 */
public abstract class Value
{
  private Type _type;

  public Value(Type type)
  {
    _type = type;
  }

  public Type getType()
  {
    return _type;
  }

  /**
   * If this value is a sequence containing exactly one element, that element
   * is returned.  (If that element is also a sequence containing exactly one element,
   * this method is applied recursively.)  Otherwise, returns this value itself.
   * <p>
   * This method should be called in cases where an operator does not accept sequence
   * values, in order to implement the requred conversions specified in the LRM.
   *
   * @return
   */
  public Value convertSequence()
  {
    return this; //to be overridden in SequenceValue
  }
```

```java
  /**
   * Attempts to coerce this Value to a boolean using the value conversions
   * specified in the Spaniel LRM.
   * <p>
   *
   * If this value is not a sequence, it is itself returned.  If this value is a
   * sequence containing exactly one element and that element is of type Boolean, that
   * element is returned.  Otherwise,  this method returns BooleanValue.TRUE if the
   * sequence is nonempty and BooleanValue.FALSE if it is empty.
   * <p>
   * This method should be called in cases where an operator requires a BooleanValue,
   * in order to implement the requred conversions specified in the LRM.
   *
   * @return
   */
  public Value coerceToBoolean()
  {
    return this; //to be overridden in SequenceValue
  }

  /**
   * Utility method that gets the type name corresponding to the type of
   * this value.  Returns "null" if the value is the NullValue.
   *
   * @return name of the type of this value, "null" if this is the NullValue.
   */
  public String getTypeName()
  {
    if (getType() == null)
      return "null";
    else
      return getType().getName();
  }

}
```

8.9.11 Variable.java

```java
/*
 * Created on Nov 10, 2004
 */
package edu.columbia.apl2107.spaniel.var_value;

/**
 * A Variable, which is a storage location for a value.
 * @author alally
 */
public class Variable
{
  private Value _value;
  private String _name;
  private int _id;

  /**
   * Create an unnamed variable.
   */
  public Variable()
  {
    this(null);
  }

  /**
   * Create a named variable.  The variable's name is only used for
   * debugging and error message generation purposes.
   */
  public Variable(String name)
  {
    _name = name;
    _id = sequenceNum++;
    _value = NullValue.instance();
  }
```

```
  /**
   * @return Returns the value of this variable
   */
  public Value getValue()
  {
    return _value;
  }

  /**
   * @return Returns the name of this variable, if one has been assigned.
   */
  public String getName()
  {
    return _name;
  }
  /**
   * @param value The value to set.
   */
  public void setValue(Value value)
  {
    this._value = value;
  }

  public String toString()
  {
    return (_name == null ? "var" : _name) + "@" + _id;
  }

  private static int sequenceNum = 0;
}
```

8.10 Spaniel Test Programs for JUnit Test Case

8.10.1 Annotation.spl

```
/* Test program for annotations, intended to be run as part of JUnit test case.
   This program expects to be passed the document "Hello World!".
*/
proc main(doc)
{
    assertEquals([0,12], doc);
    place = annotate([6,11],"Place");
    greeting = annotate([0,5],"Greeting");

    forAll(m : matching("[A-Z]",doc))
    {
      annotate(m,"Capital Letter");
    }

    assertEquals(greeting, first(subspans(doc)));
    assertEquals(place, first(rest(rest(subspans(doc)))));
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

8.10.2 ControlFlow.spl

```
/* Test program for control flow, intended to be run as part of JUnit test case. */
proc main(doc)
{
  //if - else
  foo = 42;
  if (foo > 100)
    x = "Wrong!";
  else if (bar == null && foo > 10)
```

```
    x = "Correct!";
  else
    x = "Wrong!";

  assertEquals("Correct!", x);

  if (foo > 100 || 1 == 1)
    x = "Also Correct";
  assertEquals("Also Correct", x);

  //short circuit logic
  bar = 0;
  if (foo == 42 || (bar = 1) == 1);
  assertEquals(0, bar);

  //while
  i = 1;
  j = 1;
  while(i < 10)
  {
    j = j * i;
    i = i + 1;
  }
  assertEquals(10, i);
  assertEquals(9*8*7*6*5*4*3*2, j);

  //forAll
  i = 1;
  forAll (x : oneToFive())
  {
    assertEquals(i, x);
    i = i + 1;
  }
  assertEquals(6, i);

  //break
  i = 0;
  while(true)
  {
    if (i == 10) break;
    i = i + 1;
  }
  assertEquals(10, i);

  //break from outer of nested loops
  forAll (x : oneToFive())
  {
    i = 0;
    while (i < x) { i = i + 1; }
    assertEquals(x, i);
    if (x == 4) break;
  }
  assertEquals(4, x);

  //break from inner of nested loops
  forAll (x : oneToFive())
  {
    i = 0;
    while (i < x)
    {
      i = i + 1;
      if (i == 4) break;
    }
    assertEquals(i, min(x,4));
  }
  assertEquals(5, x);
}

proc oneToFive()
{
  emit 1; emit 2; emit 3; emit 4; emit 5;
```

```
}

proc min(a,b)
{
  if (a < b)
    emit a;
  else
    emit b;
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

8.10.3 Math.spl

```
/* Test program for arithmetic, intended to be run as part of JUnit test case. */
proc main(doc)
{
  assert(5 == 5);
  assertEquals(8, 5+3);
  assertEquals(15, 5*3);
  assertEquals(10, 5+3*6/2-1-3);
  assertEquals(13, (5+3*6)/2-(1-3));
  assertEquals(2, 8%3);
  assertEquals(6,+6);
  assertEquals(7, 6--1);

  assertEquals("Five plus three is 8", "Five plus three is " + (5+3));
  assertEquals("Five plus three is 53", "Five plus three is " + 5 + 3);

  foo = 42;
  assert(foo > 10);
  assert(foo < 100);
  assert(!(foo > 100));
  assert(-foo < 0);
  assert(foo <= 42);
  assert(foo <= 43);
  assert(foo >= 0);
  assert(foo >= 42);
}

proc assert(bool)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssert", bool);
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

8.10.4 Recursion.spl

```
/* Test program for recursive procedures, intended to be run as part of JUnit test case.
*/
proc main(doc)
{
  assertEquals(1, fact(0));
  assertEquals(6, fact(3));
  assertEquals(120, fact(5));
  assertEquals(3628800, fact(10));
}

proc fact(n)
{
  if (n < 2)
    emit 1;
  else
```

```
    emit n * fact(n-1);
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

8.10.5 Sequences.spl

```
/* Test program for sequences, intended to be run as part of JUnit test case. */
proc main(doc)
{
  sequence = oneToFive(); //procs are the only way to generate sequences!
  assertEquals(1, first(sequence));
  assertEquals(2, first(rest(sequence)));
  assertEquals(3, first(rest(rest(sequence))));
  assertEquals(4, first(rest(rest(rest(sequence)))));
  assertEquals(5, first(rest(rest(rest(rest(sequence))))));
  assertEquals(empty(), first(rest(rest(rest(rest(rest(sequence)))))));

  //last first is optional, due to implicit conversion of
  //one-element sequences to primitives
  assertEquals(5, rest(rest(rest(rest(sequence)))));

  foo = twoToFive();
  bar = rest(oneToFive());
  assertEquals(foo,bar);

  //test correct conversions from sequence to boolean
  assert(oneToFive());
  assert(!empty());
  assert(!print(""));
  assert(!returnfalse());

  spans = threeSpans();
  assertEquals([1,2], first(spans));
  assertEquals([3,5], first(rest(spans)));
  assertEquals([8,9], first(rest(rest(spans))));
  assertEquals(empty(), first(rest(rest(rest(spans)))));

  //last first is optional, due to implicit conversion of
  //one-element sequences to primitives
  assertEquals([8,9], rest(rest(spans)));
}

proc oneToFive()
{
  emit 1; emit 2; emit 3; emit 4; emit 5;
}

proc twoToFive()
{
  emit 2; emit 3; emit 4; emit 5;
}

proc threeSpans()
{
  emit [1,2]; emit [3,5]; emit [8,9];
}

proc empty()
{
}

proc returnfalse()
{
  emit false;
}

proc assert(bool)
```

```
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssert", bool);
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```

8.10.6 Spans.spl

```
/* Test program for annotations, intended to be run as part of JUnit test case. */
proc main(doc)
{
    x = [1,5];
    assertEquals([1,5], x);
    assertEquals(1, x.begin);
    assertEquals(5, x.end);
    assertEquals(null, x.type);

    x.type = "Foo";
    assertEquals("Foo", x.type);
    x.type = null;
    assertEquals(null, x.type);

    x.bar = 8;
    assertEquals(8, x.bar);

    x.baz = [0,19];
    assertEquals(19, x.baz.end);
    assertEquals(null, x.baz.foo);
    x.baz.foo = 42;
    assertEquals(42, x.baz.foo);

    assertEquals([5,9], [1,9]*[5,13]);
    assertEquals([1,13], [1,9]+[5,13]);
    assertEquals([5,5], [5,5]);
    assert([2,3] > [1,3]);
    assert([1,10] < [1,9]);
}

proc assert(bool)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssert", bool);
}

proc assertEquals(a,b)
{
  javacall("edu.columbia.apl2107.spaniel.test.JUnitAssertEquals", a, b);
}
```