Spaniel

Language Reference Manual

Adam Lally
apl2107@columbia.edu
October 21, 2004

Table of Contents

1 Introduction

Spaniel (Span-based Information Extraction Language) is a 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.

1.1 Organization of this Manual

This manual is organized as follows:

Section 2 describes the lexical structure of the language, which is based on C and Java.

Section 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 4 describes the high-level structure of a Spaniel program.

Section 5 describes variables in Spaniel.

Section 6 describes statements, which are again modeled after C and Java.

Section 7 describes expressions, and defines the precedence and associativity of the language's operators.

Section 8 describes the execution of a spaniel program. Execution of a Spaniel program is unusual in one regard: procedures do not return values and abruptly terminate, but instead *emit* values and may continue to execute. Thus all procedures implicitly return sequences of values.

Section 9 describes the built-in procedures that a Spaniel runtime environment is required to provide.

Appendix A is an example of a syntactically valid Spaniel program.

1.2 Notation

1.2.1 Grammars

Syntactic structures are specified using a context-free grammar(CFG). Lexical structures are specified using regular expressions. Both use the syntax of the ANTLR language (http://www.antlr.org/). However, this manual does not contain the exact ANTLR rules used to build a Spaniel parser. Most notably, this manual does use rules with left-recursion, in order to simplify the discussion.

1.2.2 Fonts

Italics is used to indicate important terms where they are first defined.
`Monospace font` is used to indicate grammar symbols or actual source code.

2 Lexical Structure

This section specifies the lexical structure of Spaniel.

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.

2.2 Line Terminators

```
LineTerminator : ('\r' ('\n')?  | '\n' );
```

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.

2.3 Whitespace

```
WS : ( ' ' | '\t' | LineTerminator );
```

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.

2.4 Comments

```
Comment: (EndOfLineComment | BlockComment);
EndOfLineComment: '/' '/' (InputCharacter)* LineTerminator;
BlockComment: '/' '*' CommentTail;
CommentTail:
      '*' CommentTailStar |
      LineTerminator CommentTail |
      ~('*' | '\r' | '\n') CommentTail;
CommentTailStar:
      '/' | '*' CommentTailStar |
       LineTerminator CommentTail |
       ~('*' | '/' | '\r' | '\n') CommentTail;
```

As in Java, there are two types of comments. End of line comments begin with the characters `//` and end at the next `LineTerminator`. 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.

2.5 Literals

```
Literal:
      IntLiteral |
      FloatLiteral |
      StringLiteral |
      "true" |
      "false" |
      "null";
```

The strings `true` and `false` represent literal Boolean values, and `null` represents the null value. Other literals are as follows.

2.5.1 Integer Literals

```
IntLiteral: '1'..'9' (Digit)*;
Digit: ('0'..'9');
```

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.

2.5.2 Floating-point Literals

```
FloatLiteral: IntLiteral FloatTail | Fraction (Exponent)?;
FloatTail: (Fraction (Exponent)? | Exponent);
Fraction: '.' (Digit)+;
Exponent: 'e' ('+'|'-')? Digits;
```

The value of a floating-point literal is defined as the value that would be returned by the Java method Double.parseDouble when passed the text of the floating-point literal. This is a 64-bit IEEE floating-point value. It is a compile time error if a floating-point literal causes `Double.parseDouble` to throw an exception.

2.5.3 String Literals

```
StringLiteral: '"'! (~('"' | '\n' | '\r') | '"'! '"')* '"'! |
                (CharacterCode)+;
CharacterCode:
      '#'! HexDigit (HexDigit)?;
HexDigit: (Digit | 'A'..'F' | 'a'..'f');
```

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. Newline 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.

2.6 Keywords

The following character sequences are keywords:

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

2.7 Identifiers

```
ID: IdentifierStart (IdentifierChar)*;  {but not keyword or literal}
IdentifierStart: '_' | 'A'..'Z' | 'a'..'z';
IdentifierChar: IdentifierStart | '0'..'9';
```

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`, and `null`.

2.8 Separators

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

2.9 Operators

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

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: 64-bit IEEE floating point value
- String: sequence of ASCII characters
- Boolean: true or false
- Span (see section 3.1)
- Sequence (see section 3.2)

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

3.1 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.

3.2 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 8.3). Also, in Spaniel expression evaluation, sequences of a single element, in most contexts, are treated equivalently to the single element alone (see section 7.1).

4 The Structure of a Program

```
program: (procedure)+
procedure: "proc"^ ID formal_params  block;
formal_params: LPAREN! (ID (COMMA! ID)*)? RPAREN!;
```

A Spaniel *program* is one or more procedures. Each procedure has a name (identifier), a list of formal parameters (zero or more comma-separated identifiers), and a block (the body of the procedure). A block is zero or more statements enclosed in curly braces. The syntax and emantics of blocks and statements are described in section 5.

Any 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.

The details of procedure execution are described in section 8.

5 Variables

A *variable* is a storage location that can take a value of any of the types listed in section 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*: a local variable is created whenever the execution of an expression encounters an identifier that is (a) not known to be the name of an existing variable and (b) is not the initial identifier within a procedure call expression (section 7.3.3), and thus known to be the name of a procedure. 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. A field variable is created whenever a field is referenced within an lvalue expression (section 0), 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 (as described in section 8).

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.

6 Statements

```
statement:
      block |
      expression_statement SEMI! |
      if_stmt |
      forAll_stmt |
      while_stmt |
      break_stmt SEMI! |
      emit_stmt SEMI! |
      SEMI! /* empty statement */;
```

6.1 Block

```
block: LBRACE! (statement)* RBRACE!;
```

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.

6.2 Expression Statement

```
expression_statement:
    assignment SEMI! |
    proc_call_exp SEMI!;
```

Two types of expressions – assignments (section 7.9) and procedure calls (section 7.3.3) – can be used as statements. The expression is evaluated, causing any side effects it may have to take place.

6.3 If Statement

```
if_stmt:
      "if" LPAREN! expression RPAREN! statement ("else"! statement)?;
```

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.

6.4 ForAll Statement

```
forAll_stmt: "forAll"^ LPAREN! ID COLON! expression RPAREN! 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 evalutes, 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) Goto step 1.

6.5 While Statement

```
while_stmt: "while" LPAREN! expression RPAREN! 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) Goto step 1.

6.6 Break Statement

```
break_stmt: "break"^;
```

When a break statement executes, it causes execution of the immediately enclosing forAll or while statement to immediately complete.

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

6.7 Emit Statement

```
emit_stmt: "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. See section 8 for details on procedure execution.

6.8 Empty Statement

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

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 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. The only exception is *lvalue expressions* (section 0), which evaluate to variables.

7.1 Value Conversions

In Spaniel expression evaluation, sequences of a single element, in most contexts, are treated equivalently to the single element alone. Specifically, the following rule applies to the expression specifications below:

> If the expression specification does not explicitly state how sequence values are treated, then any sequence value consisting of one exactly one element is automatically converted to that one element during evaluation of the expression.

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

7.2 Lvalue Expressions

```
lvalue_exp : ID (DOT lvalue_exp)?;
```

An *lvalue expression* is an identifier, optionally followed by a dot followed by another lvalue expression. 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 a single identifier evaluates to the variable whose name is that identifier (see section 5). An lvalue expression of the form `ID DOT lvalue_exp` is evaluated by first evaluating the `lvalue_exp` after 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 named ID in span v.

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, separate by dots, thus for example `proc(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 Spaniels always return sequences, which are not lvalues.

In the remainder of this specification, unless otherwise specified, 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.

7.3 Primary Expressions

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

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

7.3.1 Literals

The syntax for literals is defined in section 2.5.

Integer, Floating-point, and String Literals evaluate to the values defined in section 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.

7.3.2 Span Expressions

```
span_exp: LBRACKET! expression COMMA! expression RBRACKET!
```

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 that 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.

7.3.3 Procedure Calls

```
proc_call_exp: ID actual_params
actual_params: LPAREN! (expression (COMMA! expression)*)? RPAREN!;
```

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 9) or declared within the user's program (see section 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 are evaluated, in order. Then the named procedure is invoked, with actual parameters equal to the values to which the sub-expressions evaluated. The details of procedure invocation are specified in section 8. The procedure call expression evaluates to the value returned by this invocation, which is always of type Sequence.

7.3.4 Parenthesized Expressions

An expression enclosed in parentheses evaluates to the same variable or value (including a sequence value) that the enclosed expression evaluates to.

7.4 Unary Expressions

```
unary_exp: PLUS^ unary_exp | MINUS^ unary_exp | NOT^ unary_exp |
           primary_exp ;
```

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. For purposes of the grammar, a primary expression is also considered to be a unary expression.

When a unary expression is evaluated, its sub-expression is first evaluated. The value to which the sub-expression evaluates is called the *operand.*

In a unary expression containing a + operator, 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.

In a unary expression containing a – operator, 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.

In a unary expression containing a ! operator, the operand must evaluate to type Boolean or type Sequence, 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). For a Sequence-typed operand, the unary expression evaluates to true if the sequence is empty, and false if it is not empty.

7.5 Multiplicative Expressions

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

The multiplicative operators are *, /, and %. They associate to the left. For purposes of the grammar, a unary expression is also considered a multiplicative expression. For multiplicative expressions that do contain an operator, evaluation is as follows:

If both operands evaluate to integers, then the expression evaluates to the result of performing integer multiplication(*), division(/), or modulo(%) 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(*) or division(/) on the operand values. Use of the operator % in this case is a run-time error.

If both operands evaluate to spans, then for the * operator 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.

7.6 Additive Expressions

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

The additive operators are + and -. They associate to the left. For purposes of the grammar, a multiplicative expression is also considered an additive expression.
For additive expressions that do contain an operator, evaluation is as follows:

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

If both operands evaluate to spans, then for the + operator, 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)]. Use of the – operator in this case is a run-time error.

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

7.7 Relational Expressions

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

The relational operators are ==, !=, <, <=. >, and >=. They do not associate, and always evaluate to a value of type Boolean. For purposes of this grammar, additive expressions are also considered relational expressions. For relational expressions that do contain an operator, evaluation is as follows:

An expression using the == operator evaluates to true if both of its values 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 eual. For sequences, equal means that all elements of the first sequence equal all elements of the second sequence, in order.

An expression using the != operator 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.

An expression using the < operator 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.

An expression using the <= operator evaluates to true if either < or == would have evaluated to true, and false otherwise.
An expression using the > operator 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.

An expression using the >= operator evaluates to true if either > or == would have evaluated to true, and false otherwise.

7.8 Conditional Expressions

Conditional expressions (both && and || expressions) in Spaniel require operands that evaluate to either Booleans or sequences. If an operand evaluates to an empty sequence, that operand is considered to have evaluated to false in the descriptions that follow. If an operand evaluates to a non-empty sequence, that operand is considered to have evaluated to true in the descriptions that follow. If either operand evaluates to a value of a type other than Boolean or Sequence, it is a run-time error.

7.8.1 And Expressions

```
and_exp: and_exp AND rel_exp |
         rel_exp;
```

The and operator is &&. It is left associative. For the purposes of the grammar, a relational expression is also considered an And expression.

When an And expression is evaluated, its first operand is immediately evaluated. If it 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. The And expression then evaluates to the same value as the second operation expression.

7.8.2 Or Expressions

```
or_exp: or_exp AND and_exp |
        and_exp;
```

The or operator is ||. It is left associative. For the purposes of the grammar, an and expression is also considered an Or expression.

When an Or expression is evaluated, its first operand is immediately evaluated. If it 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. The Or expression then evaluates to the same value as the second operation expression.

7.9 Assignment Expressions

```
assign_exp: assignment | or_exp;
assignment: lvalue_exp ASSIGN^ assign_exp;
```

An assignment expression is an assignment or an or expression. An assignment if an lvalue expression followed by the assignment operator, =, followed by an assignment expression. The = operator is right associative.

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.

7.10 Expressions

```
expression: assign_exp;
```

Where it appears elsewhere in this grammar, an expression means any assignment expression.

8 Execution

8.1 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. (See the white paper *Spaniel – Span Based Information Extraction Language*, for a simple example.)

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). 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."

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 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 9, as well as arbitrary output generated by calls into Java, which are made using the `javacall` procedure also defined in section 9.

8.2 Startup

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

1. The Spaniel program's source code is parsed, building an intermediate representation 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 is in whatever format the interpreter has specified, 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.

8.3 Procedure Invocation and Execution

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 7.3.3) 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.

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.

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.

9.1 *first*

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

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.

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 interpret should convert the value to a string in whatever way it chooses. The empty sequence is returned.

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 interpret should convert the value to a string in whatever way it chooses. The empty sequence is returned.

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).

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).

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]);
  }
}
```

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 defition of <= for spans in section 7.7.)

9.9 instancesOf

If `s` is a span, `instancesOf(s,t)` 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(s,t)
{
  forAll(x : subspans(s))
  {
    if (x.type == t)
      emit x;
  }
}
```

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.

`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 8.3.

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

Appendix A – Sample Program

The following is a syntactically valid Spaniel program. The intent of this program is to tokenizer text, then detect sentence boundaries, then detect phone numbers, an finally to detect mentions of phone calls between two phone numbers, using a very simple rule. Note that this program has not been verified to be semantically correct.

```
/*
 * Spaniel Example Program
 * Phone Number and Phone Call Detector
 */

/* Tokenize text */
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);
  }
}

/* Annotate sentences */
proc sentences(tokenSequence)
{
  start = null;
  //iterate over token sequence
  forAll (x: tokenSequence)
  {
    if (start == null)
      start = x;
    //sentences are ended by a ., ?, or ! character
    if (x.tokenType == "Punctuation" &&
        matching("[.?!]",x))
    {
      emit (annotate ([start.begin, x.end],"Sentence"));
      start = null;
    }
  }
}

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

/* Annotate Phone Calls */
proc phoneCalls(span)
{
   forAll(num1 : instancesOf("PhoneNumber"))
   {
      forAll (num2 : instancesOf("PhoneNumber",[num1.end,span.end]))
      {
        if (matching("from",[span.begin, num1.begin]) &&
            matching("to",[num1.end, num2.begin]))
        {
          emit(annotate([sentence.begin, n2.end], "PhoneCall"));
        }
      }
    }
}

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