Daffodil Extensions of DFDL

Contents

1 Introduction

Daffodil provides extensions to the DFDL specification. These functions and properties are in the namespace defined by the URI `http://www.ogf.org/dfdl/dfdl-1.0/extensions` which is normally bound to the `dfdlx` prefix like so:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/"
        xmlns:dfdlx="http://www.ogf.org/dfdl/dfdl-1.0/extensions"
>
```

The DFDL language extensions described below have Long Term Support (LTS) in Daffodil going forward, and are proposed for inclusion in a future revision of the DFDL standard. DFDL schema authors can depend on the features and behaviors defined here without fear that these extensions will be withdrawn in the future.

2 Expression Functions

2.1 `daf:error(messageArg)`

A function that can be used in DFDL expressions. This functions does not return a value or accept any arguments. When called, it causes a Parse Error or Unparse Error.

```
*This function is deprecated as of Daffodil 2.0.0.
 Use the ``fn:error(...)`` function instead.*
```

2.2 `dfdlx:trace($value, $label)`

A function that can be used in DFDL expressions, similar to the `fn:trace()` function. This logs the string `$label` followed by `$value` converted to a string and returns `$value`. The second argument must be of type `xs:string`.

2.3 `dfdlx:lookAhead(offset, bitSize)`

Read `bitSize` bits, where the first bit is located at an `offset` (in bits) from the current location. The result is a `xs:nonNegativeInteger`. Restrictions:

- offset >=0
- bitSize >= 1
- distance + bitSize <= Implementation defined limit no less than 512 bits
- Cannot be called during unparse
- Parse Error if the offset results in attempting to look ahead past EOF
- Undefined behavior if the offset results in attempting to look past the current data limit of a `dfdl:lengthKind="explicit"` surrounding element.
- The `dfdl:bitOrder` and `dfdl:byteOrder` are determined by the current schema component and data location.
- DFDL property changes between the current location and the location containing the data being read will not be used.

2.3.1 Examples of `dfdlx:lookAhead`

The following two elements both populate element `a` with the value of the next 3 bits as an unsignedInt. They are not completely equivalent because the first will consume 3 bits of the input stream where the second will not advance the input stream.

```
<xs:element name="a" type="xs:unsignedInt" dfdl:length="3"
   dfdl:lengthUnits="bits" />

<xs:element name="a" type="xs:unsignedInt" dfdl:inputValueCalc="{
   dfdlx:lookAhead(0,3) }" />
```

The following example demonstrates using lookAhead to branch based on a field in the future. In this case the choice of elements `a` vs. `b` depends on the value of the `tag` field which is found after fields `a` and `b`:

```
<xs:choice dfdl:choiceDispatchKey="{ dfdlx:lookAhead(16,8) }">
  <xs:element name="a" type="xs:int" dfdl:length="16" dfdl:choiceBranchKey
     ="1"/>
  <xs:element name="b" type="xs:int" dfdl:length="16" dfdl:choiceBranchKey
     ="2"/>
</xs:choice>
<xs:element name="tag" type="xs:int" dfdl:length="8" />
```

2.4 Bitwise Functions: `bitAnd`, `bitOr`, `bitXor`, `bitNot`, `leftShift`, `rightShift`

These functions are defined on types `long`, `int`, `short`, `byte`, `unsignedLong`, `unsignedInt`, `unsignedShort`, and `unsignedByte`

2.4.1 `dfdlx:bitAnd(arg1, arg2)`

This computes the bitwise AND of two integers.

- Both arguments must be signed, or both must be unsigned.
- If the two arguments are not the same type the smaller one is converted into the type of the larger one.
- If the smaller argument is signed, this conversion does sign-extension.
- The result type is the that of the largest argument.

2.4.2 `dfdlx:bitOr(arg1, arg2)`

This computes the bitwise OR of two integers.

- Both arguments must be signed, or both must be unsigned.
- If the two arguments are not the same type the smaller one is converted into the type of the larger one.
- If the smaller argument is signed, this conversion does sign-extension.
- The result type is the that of the largest argument.

2.4.3 `dfdlx:bitXor(arg1, arg2)`

This computes the bitwise Exclusive OR of two integers.

- Both arguments must be signed, or both must be unsigned.
- If the two arguments are not the same type the smaller one is converted into the type of the larger one.
- If the smaller argument is signed, this conversion does sign-extension.
- The result type is the that of the largest argument.

2.4.4 `dfdlx:bitNot(arg)`

This computes the bitwise NOT of an integer. Every bit is inverted. The result type is the same as the argument type.

2.4.5 `dfdlx:leftShift(value, shiftCount)`

This is the *logical* shift left, meaning that bits are shifted from less-significant positions to more-significant positions.

- The left-most bits shifted out are discarded.
- Zeros are shifted in for the right-most bits.
- The result type is the same as the `value` argument type.
- It is a processing error if the `shiftCount` argument is < 0.
- It is a processing error if the `shiftCount` argument is greater than the number of bits in the type of the value argument.

2.4.6 `dfdlx:rightShift(value, shiftCount)`

This is the *arithmetic* shift right, meaning bits move from most-significant to less-significant positions. If *logical* (zero-filling) shift right is needed, you must use unsigned types.

- The `value` argument is shifted by the `shiftCount`.
- The right-most bits shifted out are discarded.
- If the `value` is signed, then the sign bit is shifted in for the left-most bits.
- If the `value` is unsigned, then zeros are shifted in for the left-most bits.
- The result type is the same as the `value` argument type.
- It is a processing error if the `shiftCount` argument is < 0.
- It is a processing error if the `shiftCount` argument is greater than the number of bits in the type of the value argument.

2.5 `dfdlx:doubleFromRawLong(longArg): double` and `dfdlx:doubleToRawLong(doubleArg): long`

IEEE binary float and double values that are not NaN will parse to base 10 text and unparse back to the same exact IEEE binary bits. However, the same cannot be said for NaN (not a number) values, of which there are many bit patterns. To preserve float and double NaN values bit for bit you can use these functions to compute `xs:long` values

that enable the DFDL Infoset to preserve the bits of a float or double value even if it is a NaN.

3 Properties

3.1 `dfdlx:parseUnparsePolicy`

A property applied to simple and complex elements, which specifies whether the element supports only parsing, only unparsing, or both parsing and unparse. Valid values for this property are `parse`, `unparse`, or `both`. This allows one to leave off properties that are required for only parse or only unparse, such as `dfdl:outputValueCalc` or `dfdl:outputNewLine`, so that one may have a valid schema if only a subset of functionality is needed.

All elements must have a compatible parseUnparsePolicy with the compilation parseUnparsePolicy (which is defined by the root element daf:parseUnparsePolicy and/or the Daffodil parseUnparsePolicy tunable) or it is a Schema Definition Error. An element is defined to have a compatible parseUnparsePolicy if it has the same value as the compilation parseUnparsePolicy or if it has the value `both`.

For compatibility, if this property is not defined, it is assumed to be `both`.

3.2 `dfdlx:layer`

Layers provide algorithmic capabilities for decoding/encoding data or computing checksums. Some are built-in to Daffodil. New layers can be created in Java/Scala and plugged-in to Daffodil dynamically. There is separate Layer documentation.

3.3 `dfdlx:direction`

This property has

3.4 `dfdlx:repType`, `dfdlx:repValues`, and `dfdlx:repValueRanges`

TBD

4 Extended Behaviors

4.1 Type `xs:hexBinary`

Daffodil allows `dfdlx:lengthUnits='bits'` for this simple type.