# DataScript Language Overview

Harald Wellmann <HWellmann@harmanbecker.com>

Revision History

Revision 0.1 20 July 2006

Initial version.

Revision 0.2 16 November 2006

Added section on Comments.

Revision 0.3 3 December 2006

Added section on Packages and Imports.

Revision 0.4 11 January 2007

Added section on Subtypes.

Revision 0.5 23 May 2007

More details on packages. Scope of enum items changed from global to local. More background in introduction.

Revision 0.6 28 June 2007

New sum operator. Functions. New section on Relational Extensions.

Revision 0.7 29 November 2007

New choice type. Functions may return compound types. Explicit parameters. bitsizeof operator. align(n) modifier. UTF-8 string encoding.

## Table of Contents

1. Introduction ........ 2
   1.1. Motivation ........ 2
   1.2. History and Background ........ 2
2. Literals ........ 3
3. Base Types ........ 4
   3.1. Integer Base Types ........ 4
   3.2. Bit Field Types ........ 4
   3.3. String Types ........ 4
4. Enumeration Types ........ 4
5. Compound Types ........ 5
   5.1. Sequence Types ........ 5
   5.2. Union Types ........ 5
   5.3. Choice Types ........ 6
   5.4. Constraints ........ 7
   5.5. Optional Members ........ 7
   5.6. Functions ........ 7
6. Array Types ........ 8
   6.1. Fixed and Variable Length Arrays ........ 8

6.2. Implicit Length Arrays ........ 8
7. Labels, Offsets and Alignment ........ 8
7.1. Labels and Byte Offsets ........ 8
7.2. Alignment and Padding ........ 9
8. Expressions ........ 9
8.1. Binary Operators ........ 10
8.2. Unary Operators ........ 10
8.3. Ternary Operators ........ 11
8.4. Operator Precedence ........ 12
9. Nested Types ........ 12
10. Member Access and Contained Types ........ 13
11. Parameterized Types ........ 14
12. Subtypes ........ 15
13. Comments ........ 15
13.1. Standard Comments ........ 15
13.2. Documentation Comments ........ 16
14. Packages and Imports ........ 16
14.1. Type Name Visibility ........ 17
14.2. Packages and Files ........ 17
15. Relational Extensions ........ 18
15.1. Motivation ........ 18
15.2. SQL Tables ........ 18
15.3. SQL Databases ........ 20
15.4. SQL Integers ........ 20
References ........ 20

# 1. Introduction

## 1.1. Motivation

There are dozens of languages for modelling abstract datatypes. For some of these, the binary representation of the defined types is implementation dependent and of no concern. Others do provide a binary encoding, but there is usually no way to retrofit an abstract specification to an existing binary format.

DataScript is a formal language for modelling binary datatypes, bitstreams or file formats. Using a formal language for defining such binary datatypes resolves all ambiguities typically found in textual or tabular specifications.

In addition, one can automatically generate encoders and decoders for a given binary format from such a formal specification, so that application developers do not have to worry about serialization and can focus on application logic instead.

The present document describes a DataScript dialect supported by our implementation called `rds` (short for Relational DataScript), which is available from SourceForge [rds] under a BSD License. This Language Overview is more of a User's Manual than a formal language specification.

## 1.2. History and Background

DataScript was designed by Godmar Back [Back]. His reference implementation of a DataScript compiler also includes a Java code generator producing Java classes which are able to read and write a binary stream that complies with a DataScript specification.

`rds` and the DataScript dialect defined in this document are a spin-off of the *Physical Storage Format Standardization Initiative* (PSI), a joint effort of about 20 partner companies in the automotive industry to standardize a car navigation database format.

After evaluating different data modelling languages and toolsets, including ASN.1, CSN.1, UML, XML Schema and others, DataScript was selected by the PSI as the closest match to our requirements.

While Back's reference implementation [DataScript] provided a great start, we found that some language extensions were desirable to better support our specific requirements. For this reason, we branched off our own DataScript Tools or `dstools` project from Back's reference implementation.

As a major addition to the DataScript language, we introduced relational extensions, which permit the definition of hybrid data models, where the high-level access structures are implemented by relational tables and indices, whereas the bulk data are stored in single columns as BLOBs with a format defined in DataScript, hence the name *Relational DataScript*. `rds` is currently built on top of the SQLite embedded database [SQLite].

## 2. Literals

The DataScript syntax for literal values is similar to the Java syntax. There are no character literals, only string literals with the usual escape syntax. Integer literals can use decimal, hexadecimal, octal or binary notation.

Examples:

- Decimal: `100, 4711, 255`
- Hexadecimal: `0xCAFEBABE, 0Xff`
- Octal: `044, 0377`
- Binary: `111b, 110b, 001B`
- String: `"You"`

Hexadecimal digits and the `x` prefix as well as the `b` suffix for binary types are case-insensitive.

String literals correspond to zero-terminated UTF-8-encoded strings. Thus, the literal `"You"` corresponds to a sequence of 4 bytes equal to the binary representation of the integer literal `0x596F7500`. Other character encodings (e.g. ISO 8859-1 or UTF-16) are not supported.

# 3. Base Types

## 3.1. Integer Base Types

DataScript supports the following integer base types

- Unsigned Types: `uint8, uint16, uint32, uint64`
- Signed Types: `int8, int16, int32, int64`

These types correspond to unsigned or signed integers represented as sequences of 8, 16, 32 or 64 bits, respectively. Negative values are represented in two's complement, i.e. the hex byte `FF` is 255 as `uint8` or -1 as `int8`.

The default byte order is big endian. Thus, for multi-byte integers, the most significant byte comes first. Within each byte, the most significant bit comes first.

Example: The byte stream `02 01` (hex) interpreted as `int16` has the decimal value 513. As a bit stream, this looks like `0000 0010 0000 0001`. Bit 0 is `0`, bit 15 is `1`.

## 3.2. Bit Field Types

A bit field type is denoted by `bit:1, bit:2`, ... The colon must be followed by a positive integer literal, which indicates the length of the type in bits. The length is not limited. A bit field type corresponds to an unsigned integer of the given length. Thus, `bit:16` and `uint16` are equivalent.

Signed bit field types are not supported.

Variable length bit field types can be specified as `bit<`*expr*`>`, where *expr* is an expression of integer type to be evaluated at run-time.

## 3.3. String Types

A string type is denoted by `string`. It is represented by a sequence of bytes in UTF-8 encoding, terminated by a zero byte. Thus, the encoded size of a string with *n* characters is at least *n*+1 bytes, and it may even be larger if some of the characters have a multibyte UTF-8 encoding.

Since DataScript models arbitrary bitstreams, the term *byte* should not be taken too literally in this context: A byte in this sense is a group of 8 successive bits, where the offset of the first bit of each group from the enclosing type or the beginning of the stream is not necessarily divisible by 8.

# 4. Enumeration Types

An enumeration type has a base type which is an integer type or a bit field type. The members of an enumeration have a name and a value which may be assigned explicitly or implicitly. A member that does not have an initializer gets assigned the value of its predecessor incremented by 1, or the value 0 if it is the first member.

```
enum bit:3 Color
{
    NONE  = 000b,
    RED   = 010b,
    BLUE,
    BLACK = 111b
};
```

In this example, `BLUE` has the value 3. When decoding a member of type `Color`, the decoder will read 3 bits from the stream and report an error when the integer value of these 3 bits is not one of 0, 2, 3 or 7.

An enumeration type provides its own lexical scope, similar to Java and dissimilar to C++. The member names must be unique within each enumeration type, but may be reused in other contexts with different meanings. Referring to the example, any other enumeration type `Foo` may also contain a member named `NONE`.

In expressions outside of the defining type, enumeration members must always be prefixed by the type name and a dot, e.g. `Color.NONE`.

## 5. Compound Types

### 5.1. Sequence Types

A sequence type is the concatenation of its members. There is no padding or alignment between members. Example:

```
MySequence
{
    bit:4    a;
    uint8    b;
    bit:4    c;
};
```

This type has a total length of 16 bits or 2 bytes. As a bit stream, bits 0-3 correspond to member `a`, bits 4-11 represent an unsigned integer `b`, followed by member `c` in bits 12-15. Note that member `b` overlaps a byte boundary, when the entire type is byte aligned. But `MySequence` may also be embedded into another type where it may not be byte-aligned.

### 5.2. Union Types

A union type corresponds to exactly one of its members, which are also called branches.

```
union VarCoordXY
{
    CoordXY8    coord8  : width == 8;
    CoordXY16   coord16 : width == 16;
    CoordXY24   coord24 : width == 24;
    CoordXY16   coord32 : width == 32;
};
```

In this example, the union `VarCoordXY` has two branches `coord8` and `coord16`. The syntax of a member definition is the same as in sequence types. However, each member should be followed by a constraint. This is a boolean expression introduced by a colon. The terms involved in the constraint must be visible in the scope of the current type at compile time and must have been decoded at runtime before entering the branch.

The decoding semantics of a union type is a trial-and-error method. The decoder tries to decode the first branch. If a constraint fails, it proceeds with the second branch, and so on. If all branches fail, a decoder error is reported for the union type.

A branch without constraints will never fail, so any following branches will never be matched. This can be used to express a default branch of a union, which should be the last member.

When all constraints of a union depend on the same member, a choice type is usually more convenient.

## 5.3. Choice Types

A choice type is a shorthand notation for a union where all constraints compare a given member or parameter to one or more constant values.

```
choice VarCoordXY on Coord.width
{
    case  8: CoordXY8  coord8;
    case 16: CoordXY16 coord16;
    case 24: CoordXY24 coord24;
    case 32: CoordXY32 coord32;
};
```

A choice type depends on a *selector expression* following the `on` keyword. Each branch of the choice type is preceded by one or more case labels with a literal value. After evaluating the selector expression, the decoder will directly select the branch labelled with a literal value equal to the selector value. This is more efficient than the trial-and-error method applied to union types. Loosely speaking, a union type corresponds to a chain of `if ... else if ... else if ... else` statements, whereas a choice type is equivalent to a single `switch` statement.

In the example above, the selector expression refers to a member `width` of a type named `Coord` containing the current choice type (see Section 10, “Member Access and Contained Types”). Alternatively, the selector may be a parameter of the choice type (see Section 11, “Parameterized Types”):

```
choice VarCoordXY(uint8 width) on width
{
    case  8: CoordXY8  coord8;
    case 16: CoordXY16 coord16;
    case 24: CoordXY24 coord24;
    case 32: CoordXY32 coord32;
};
```

A given branch of a choice may have more than one case label. In this case, the branch is selected when the selector value is equal to any of the case label values. A choice type may have a default branch which is selected when no case label matches the selector value. The decoder will throw an exception when there is no default branch and the selector does not match any case label. Any branch, including the default branch, may be empty, with a terminating semicolon directly following the label. It is good practice to insert a comment in this case. When the selector expression has an enumeration type, the enumeration type prefix may be omitted from the case label literals.

```
choice AreaAttributes(AreaType type) on type
{
    case AreaType.COUNTRY:    // The prefix "AreaType." is optional
    case STATE:
    case CITY:
        RegionAttributes regionAttr;

    case MAP:
        /* empty */ ;

    case ROAD:
        RoadAttributes roadAttr;

    default:
        DefaultAttributes defaultAttr;
};
```

## 5.4. Constraints

A constraint may be specified for any member of a compound type, not just for selecting a branch of a union. In a sequence type, after decoding a member with a constraint, the decoder checks the constraint and reports an error if the constraint is not satisfied.

There is a shorthand syntax for a constraint that tests a field for equality. *Type fieldName* `=` *expr*`;` is equivalent to *Type fieldName* `:` *fieldName* `==` *expr*`;`

## 5.5. Optional Members

A sequence type may have optional members:

```
ItemCount
{
    uint8    count8;
    uint16   count16  if count8 == 0xFF;
};
```

An optional member has an `if` clause with a boolean expression. The member will be decoded only if the expression evaluates to true at run-time.

Optional members are a more compact and convenient alternative to a union with two branches one of which is empty.

## 5.6. Functions

A compound type may contain functions:

```
ItemCount
{
    uint8    count8;
    uint16   count16  if count8 == 0xFF;

    function uint16 getValue()
    {
        return (count8 == 0xFF) ? count16 : count8;
    }
};
```

The return type of a function has to be a standard integer or compound type, and the function parameter list must be empty. The function body may contain nothing but a return statement with an expression matching the return type.

Functions are intended to provide no more than simple expression semantics. There are no plans to add more advanced type conversion or even procedural logic to

DataScript. (For complex logic, it would be more sensible to bind native functions to DataScript.)

## 6. Array Types

### 6.1. Fixed and Variable Length Arrays

An array type is like a sequence of members of the same type. The element type may be any other type, except an array type. (Two dimensional arrays can be emulated by wrapping the element type in a sequence type.)

The length of an array is the number of elements, which may be fixed (i.e. set at compile-time) or variable (set at run-time). The elements of an array have indices ranging from 0 to *n*-1, where n is the array length.

The notation for array types and elements is similar to C:

```
ArrayExample
{
    uint8    header[256];
    int16    numItems;
    Element  list[numItems];
};
```

`header` is a fixed-length array of 256 bytes; list is an array with *n* elements, where *n* is the value of `numItems`. Individual array elements may be referenced in expressions with the usual index notation, e.g. `list[2]` is the third element of the `list` array.

Constraints on all elements of an array can be expressed with the `forall` operator, see Section 8.3.2, “Quantified Expression”.

### 6.2. Implicit Length Arrays

An array type may have an implicit length indicated by an empty pair of brackets. In this case, the decoder will continue matching instances of the element type until a constraints fail or the end of the stream is reached.

```
ImplicitArray
{
    Element    list[];
};
```

The length of the list array can be referenced as `lengthof list`, see Section 8.2.5, “lengthof Operator”.

## 7. Labels, Offsets and Alignment

### 7.1. Labels and Byte Offsets

The name of a member of integral type may be used as a label on another member to indicate its byte offset in the enclosing sequence:

```
Tile
{
    TileHeader    header;
    uint32        stringOffset;
    uint16        numFeatures;

stringOffset:
    StringTable   stringTable;
};
```

In this example, the byte offset of member `stringTable` from the beginning of the `Tile` instance is given by the value of `stringOffset`.

The offset of a label is relative to the enclosing sequence by default. If the offset is relative to some other type containing the current one, this is indicated by a global label, where the type name is used as a prefix, followed by a double colon:

```
Database
{
    uint32    numTiles;
    Tile      tiles[numTiles];
};

Tile
{
    TileHeader    header;
    uint32        stringOffset;
    uint16        numFeatures;

Database::stringOffset:
    StringTable   stringTable;
};
```

## 7.2. Alignment and Padding

Since labels always refer to byte offsets, a given member within a sequence type cannot be labelled if it is not guaranteed to be byte-aligned. To overcome this restriction, an alignment can be specified explicitly:

```
Tile
{
    TileHeader    header;
    uint32        stringOffset;
    uint16        numBits;
    bit:1         bits[numBits];

align(8):
stringOffset:
    StringTable   stringTable;
};
```

The align(n) modifier causes the decoder to skip 0..n-1 bits so that the bit offset from the beginning of the stream is divisible by n. n may be any integer literal. Alignment modifiers may be used in any sequence type, independent of labels:

```
AlignmentExample
{
    bit:11       a;

align(32):
    uint32       b;
};
```

The size of the AlignmentExample type is 64 bits; without the alignment modifier, the size would be 43 bits.

# 8. Expressions

The semantics of expression and the precedence rules for operators is the same as in Java, except where stated otherwise. DataScript has a number of special operators `sizeof`, `lengthof`, `is` and `forall` that will be explained in detail below.

The following Java operators have no counterpart in DataScript: `++`, `--`, `>>>`, `instanceof`.

## 8.1. Binary Operators

### 8.1.1. Arithmetic Operators

The integer arithmetic operations include `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo). In addition, there are the shift operators `<<` and `>>`.

### 8.1.2. Relational Operators

There are the following relational operators for integer expressions: `==` (equal to), `!=` (not equal to), `<` (less than), `<` (less than or equal), `>` (greater than), `>=` (greater than or equal).

The equality operators `==` and `!=` may be applied to any type

### 8.1.3. Boolean operators

The boolean operators `&&` (and) and `||` (or) may be applied to boolean expressions.

### 8.1.4. Bit operators

The bit operators `&` (bitwise and), `|` (bitwise or), `^` (bitwise exclusive or) may be applied to integer types.

### 8.1.5. Assignment operators

The assignment operator `=` and the combined assignment operators `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=` have the usual semantics.

### 8.1.6. Comma operator

The comma operator `,` evaluates to the expression on the right hand side.

### 8.1.7. Postfix operators

The postfix operators include `[]` (array index), `()` (instantiation with argument list or function call), and `.` (member access).

## 8.2. Unary Operators

### 8.2.1. Boolean Negation

The negation operator `!` is defined for boolean expressions.

### 8.2.2. Integer operators

For integer expressions, there are `+` (unary plus), `-` (unary minus) and `~` (bitwise complement).

### 8.2.3. sizeof Operator

The `sizeof` operator returns the size of a type or an expression in bytes. `sizeof` may not be used, when the size in bits is not divisible by 8. When `sizeof` is applied to a type name, the size of the type must be fixed and known at compile time. When `sizeof` is a applied to a member, it refers to the actual size of the member after decoding.

### 8.2.4. bitsizeof Operator

The `bitsizeof` operator returns the size of a type or an expression in bits. When `bitsizeof` is applied to a type name, the size of the type must be fixed and known at compile time. When `bitsizeof` is a applied to a member, it refers to the actual size of the member after decoding.

### 8.2.5. lengthof Operator

The `lengthof` operator may be applied to an array member and returns the actual length (i.e. number of elements of an array.Thus, given `int32 a[5]`, the expression `lengthof a` evaluates to 5. This is not particularly useful for fixed or variable length arrays, but it is the only way to refer to the length of an implicit length array.

### 8.2.6. is Operator

The `is` operator can be applied to two field names, e.g. `x is y`. `x` must be a member of union type, and `y` must be one of the branch names of that union. The expression is true if and only if the decoder has selected branch y for the union.

### 8.2.7. sum Operator

The `sum` operator is defined for arrays with integer element type (this includes bit fields). `sum(a)` evaluates to the sum of all elements of the array `a`.

## 8.3. Ternary Operators

### 8.3.1. Conditional Expression

A conditional expression *booleanExpr* ? *expr1* : *expr2* has the value of *expr1* when *booleanExpr* is true. Otherwise, it has the value of *expr2*.

### 8.3.2. Quantified Expression

A quantified expression has the form `forall` *indexIdentifier* `in` *arrayExpr* `:` *booleanExpr*. The quantified expression is true if and only if the *booleanExpr* is true for all indices of the array. This is only useful when the boolean expression after the colon involves the array expression and the index identifier from the left hand side.

Example: The constraint

```
forall i in a : (i == 0) || (a[i] == a[i-1]+1)
```

means the elements of `a` are a sequence of consecutive integers.

### 8.4. Operator Precedence

In the following list, operators are grouped by precedence in ascending order. Operators on the bottom line have the highest precedence and are evaluated first. All operators on the same line have the same precedence and are evaluated left to right, except assignment operators which are evaluated right to left.

- comma
- assignment
- `forall`
- `? :`
- `||`
- `&&`
- `|`
- `^`
- `&`
- `== !=`
- `< > <= >=`
- `<< >>`
- `+ -`
- `* / %`
- cast
- unary `+ - ~ !`
- `sizeof bitsizeof lengthof sum`
- `[] () . is`

## 9. Nested Types

DataScript syntax permits the definition of nested types, however, it is not easy to define the semantics of such types in a consistent way. For the time being, the only supported use is a sequence type definition within a sequence or union field definition, or a union type definition within a sequence field definition, and even this should be avoided in favour of a reference to a type defined at global scope. Example:

```
VarCoord
{
    uint8        width;
    union
    {
        {
            int16    x;
            int16    y;
        } coord16 : width == 16;
        {
            int32    x;
            int32    y;
        } coord32 : width == 32;
    } coord;
};
```

The sequence type `VarCoord` contains the member `coord` which has a nested union type definition. This union type has two members each of which is a nested sequence type. All nested types in this example are anonymous, but this it not necessary.

The nested type definitions can be avoided as follows:

```
VarCoord
{
    uint8    width;
    Coords   coords;
};

union Coords
{
    Coord16    coord16 : VarCoord.width == 16;
    Coord32    coord32 : VarCoord.width == 32;
};

Coord16
{
    int16    x;
    int16    y;
};

Coord32
{
    int32    x;
    int32    y;
};
```

Note that the constraints for the members of the `Coords` union refer to the containing type `VarCoord`. This is explained in more detail in the following section.

## 10. Member Access and Contained Types

The dot operator can be used to access a member of a compound type: the expression *f.m* is valid if

- *f* is a field of a compound type *C*
- The type *T* of *f* is a compound type.
- *T* has a member named *m*.

The value of the expression *f.m* can be evaluated at run-time only if the member *f* has been evaluated before.

There is a second use of the dot operator involving a type name:

At run-time, each compound type *C* (except the root type) is contained in a type *P* which has a member of type *C* which is currently being decoded. Within the scope of *C*, members of the parent type *P* may be referenced using the dot operator *P.m*.

The containment relation is extended recursively: If *C* is contained in *P* and *P* is contained in *Q*, then *Q.m* is a valid expression in the scope of *C*, denoting the member *m* of the containing type *Q*.

Example:

```
Header
{
    uint32    version;
    uint16    numItems;
};


Message
{
    Header    h;
    Item      items[h.numItems];
};

Item
{
    uint16    p;
    uint32    q if Message.h.version >= 10;
};
```

Within the scope of the `Message` type, `header` refers to the field of type `Header`, and `header.numItems` is a member of that type. Within the scope of the `Item` type, the names `h` or `Header` are not defined. But `Item` is contained in the Message type, and `h` is a member of `Message`, so `Message.h` is a valid expression of type `Header`, and `Message.h.version` references the `version` member of the `Header` type.

## 11. Parameterized Types

The definition of a compound type may be augmented with a parameter list, similar to a parameter list in a Java method declaration. Each item of the parameter list has a type and a name. Within the body of the compound type definition, parameter names may be used as expressions of the corresponding type.

To use a parameterized type as a field type in another compound type, the parameterized type must be instantiated with an argument list matching the types of the parameter list.

For instance, the previous example can be rewritten as

```
Header
{
    uint32    version;
    uint16    numItems;
};


Message
{
    Header    h;
    Item(h)   items[h.numItems];
};

Item(Header header)
{
    uint16    p;
    uint32    q if header.version >= 10;
};
```

When the element type of an array is parameterized, a special notation can be used to pass different arguments to each element of the array:

```
Database
{
    uint16                          numBlocks;
    BlockHeader                     headers[numBlocks];
    Block(headers[blocks$index])    blocks[numBlocks];
};

BlockHeader
{
    uint16 numItems;
    uint32 offset;
};

Block(BlockHeader header)
{
Database::header.offset:
    Item    items[header.numItems];
};
```

`blocks$index` denotes the current index of the `blocks` array. The use of this expression in the argument list for the `Block` reference indicates that the i-th element of the `blocks` array is of type `Block` instantiated with the i-th header `headers[i]`.

## 12. Subtypes

A subtype definition defines a new name for a given type, optionally in combination with a constraint. When the constraint is omitted, this is rather like a `typedef` in C:

```
subtype uint16  BlockIndex;


Block
{
    BlockIndex    index;
    BlockData     data;
};
```

A constraint in the subtype definition is, as usual, a boolean expression introduced by a colon which may contain the keyword `this` to refer to the current type:

```
subtype uint16 BlockIndex : 1 <= this && this < 1024;
```

Subtype constraints will be checked by the decoder for every occurrence of the given subtype in a field definition.
*Implementation note: Subtypes were introduced in version rds 0.7. Subtype constraints are not yet implemented.*

## 13. Comments

### 13.1. Standard Comments

DataScript supports the standard comment syntax of Java or C++. Single line comments start with `//` and extend to the end of the line. A comments starting with `/*` is terminated by the next occurrence of `*/`, which may or may not be on the same line.

```
// This is a single-line comment.


/* This is an example
   of a multi-line comment
   spanning three lines. */
```

### 13.2. Documentation Comments

To support inline documentation within a DataScript module, multi-line comments starting with `/**` are treated as special documentation comments. The idea and syntax are borrowed from Java(doc). A documentation comment is associated to the following type or field definition. The documentation comment and the corresponding definition may only be separated by whitespace.

```
/**
 * Traffic flow on links.
 */
enum bit:2 Direction
{
    /** No traffic flow allowed */
    NONE,
    /** Traffic allowed from start to end node. */
    POSITIVE,
    /** Traffic allowed from end to start node. */
    NEGATIVE,
    /** Traffic allowed in both directions. */
    BOTH
};
```

The content of a documentation comment, excluding its delimiters, is parsed line by line. Each line is stripped of leading whitespace, a sequence of asterisks (`*`), and more whitespace, if present. After stripping, a comment is composed of one or more paragraphs, followed by zero or more tag blocks. Paragraphs are separated by lines.

A line starting with whitespace and a keyword preceded by an at-sign (`@`) is the beginning of a tag block and also indicates the end of the preceding tag block or comment paragraph.

The only tag currently supported is `@param`, which is used for documenting the arguments of a parameterized type. The documentation comment should contain one `@param` block for each argument in the correct order. The `@param` tag is followed by the parameter name and the parameter description. The parameter name must be enclosed by whitespace.

```
/**
 * This type takes two arguments.
 * @param  arg1   The first argument.
 * @param  arg2   The second argument.
 */
ParamType(Foo arg1, Blah arg2)
{
    ...
};
```

## 14. Packages and Imports

## 14.1. Type Name Visibility

Complex DataScript specifications should be split into multiple packages stored in separate source files. Every user-defined type belongs to a unique package. For backward compatibility, there is an unnamed default package used for files without an explicit package declaration. It is strongly recommended to use a package declaration in each DataScript source file.

A package provides a lexical scope for types. Type names must be unique within a package, but a given type name may be defined in more than one package. If a type named `Coordinate` is defined in package `com.acme.foo`, the type can be globally identified by its fully qualified name `com.acme.foo.Coordinate`, which is obtained by prefixing the type name with the name of the defining package, joined by a dot. Another package `com.acme.foo.bar` may also define a type named `Coordinate`, having the fully qualified name `com.acme.bar.Coordinate`.

By default, types from other packages are not visible in the current package, unless there are imported explicitly. The package and import syntax and semantics follow the Java example.

```
package map;

import common.geometry.*;
import common.featuretypes.*;
```

Import declarations only have any effect when there is a reference to a type name not defined in the current package. If package `map` defines its own `Coordinate` type, any reference to that within package map will be resolved to the local type `map.Coordinate`, even when one or more of the imported packages also define a type named `Coordinate`.

On the other hand, if package `map` references a `Coordinate` type but does not define it, the import declarations are used to resolve that type in one of the imported packages. In that case, the referenced type must be matched by exactly one of the imported packages. It is obviously a semantic error if the type name is defined in none of the packages. It is also an error if the type name is defined in two or more of the imported packages. The order of the import declarations does not matter.

Individual types can be imported using their fully qualified name:

```
import common.geometry.Geometry;
```

This single import has precedence over any wildcard import. It prevents an ambiguity with `common.featuretypes.Geometry`. However, it would be a semantic error to have another single import of the same type name from a different package.

*Implementation note: rds 0.8 supports wildcard imports. Single imports are not yet implemented but will be added in the near future. There are currently no plans to implement references to types by their fully qualified names, as this would cause parser ambiguities with the nested type syntax, see Section 9, "Nested Types".*

## 14.2. Packages and Files

Package and file names are closely related. Each package must be located in a separate file. The above example declares a package `map` stored in a source file `map.ds`. The import declarations direct the parser to locate and parse source files `common/geometry.ds` and `common/featuretypes.ds`.

Imported files may again contain import declarations. Cyclic import relations between packages are supported but should be avoided. The DataScript parser takes care to parse each source file just once.

# 15. Relational Extensions

## 15.1. Motivation

With its basic language features presented in the previous sections, DataScript provides a rich language for modelling binary data streams, which are intended to be parsed sequentially. Direct access to members in the stream is usually not possible, except for labels specifying the offset of a given member. Navigation between semantically related members at different positions in the stream cannot be expressed at the stream level. Member insertions or updates are not supported..

All in all, the stream model is not an adequate approach for updatable databases in the gigabyte size range with lots of internal cross-references where fast access to individual members is required. In a desktop or server environment, it would be a natural approach to model such a database as a relational database using SQL.

However, in an embedded environment with limited storage space and processing resources, a full-fledged relational schema is too heavy-weight. To have the best of both worlds, i.e. compact storage on the one hand and direct member access including updates on the other hand, one can adopt a hybrid data model: In this hybrid model, the high-level access structures are strictly relational, but most of the low-level data are stored in binary large objects (BLOBs), where the internal structure of each BLOB is modelled in DataScript.

For example, we can model a digital map database as a collection of tiles resulting from a rectangular grid where the tiles are numbered row-wise. The database has a rather trivial schema:

```
CREATE TABLE europe (
  tileNum INT NOT NULL PRIMARY KEY,
  tile BLOB NOT NULL);
```

Accessing or updating any given tile can simply be delegated to the relational DBMS. Assuming that the tile BLOBs have a reasonable size, each tile can be decoded on the fly to access the individual members within the tile.

For seamless modelling of this hybrid approach, we decided to add relational extensions to DataScript. Some SQL concepts have been translated to DataScript, others are transparent to DataScript and can be embedded as literal strings to be passed to the SQL backend. Since it is hard to find a natural border between native DataScript and embedded SQL, the relational extensions of DataScript should be regarded as preliminary.

## 15.2. SQL Tables

### 15.2.1. Table Types and Instances

An SQL table type is a special case of a compound type, where the members of the type correspond to the columns of a relational table. Members of integer or string type translate to the corresponding SQL column types. Members of compound or array type correspond to BLOB columns. Members of type `uint8[n]` correspond to a `CHAR(n)` type. In Relational DataScript, we can express the above example as follows:

```
sql_table GeoMap
{
  int32   tileId sql "PRIMARY KEY";
  Tile    tile;
};


GeoMap europe;
GeoMap america;
```

It is important to note that the `GeoMap` is a table *type* and not a table. A table is defined by the instance `europe` of type `GeoMap`. Table types have no direct equivalent in SQL. They can be used to create tables with identical structure and column names. Each instance of an `sql_table` type in DataScript translates to an SQL table where the table name in SQL is equal to the instance name in DataScript. A member definition may include an SQL constraint introduced by the keyword `sql`, followed by a literal string which is passed transparently to the SQL engine.

Thus, the DataScript instance `america` gives rise to the following SQL table:

```
CREATE TABLE america (
  tileNum INT NOT NULL PRIMARY KEY,
  tile BLOB NOT NULL);
```

*Note: The current definition of sql_tables is not consistent with the abstract data type and value semantics of plain DataScript. Actually, a table instance is itself a composite type, represented by a set of rows, and each row item is an instance of the corresponding column type. To model this more adequately and to avoid embedded SQL for the primary key property, we are planning to change our definitions.*
*An SQL table is a map container type mapping (primary) keys to values: sql_map<int32, Tile> GeoMap is a container type. An instance of this type is an SQL table. A map entry is realized as a table row, corresponding to a key-value pair. Both key and value may be composed of one or more columns.*

### 15.2.2. Explicit Parameters

An `sql_table` type differs from a sequence type in that there is no decoder function that automatically reads all members of a table row. The application has to build its own queries an invoke a decoder function on each BLOB column explicitly. In the case where an `sql_table` member is an instance of a parameterized type, the application may want to derive the parameter values from the context (e.g. other table columns), which is not available to the DataScript decoder. In this case, the type arguments shall be marked with the keyword `explicit` to indicate that these values will be set explicitly be the application. Otherwise, the decoder would complain about not being able to evaluate the type arguments.

```
Tile(uint8 level, uint8 width)
```

```
{
    ...
};


sql_table TileTable
{
    uint32    tileId;
    uint32    version;
    Tile(explicit level, explicit width)   tile;
};
```

## 15.3. SQL Databases

Since an SQL table is always contained in an SQL database, we introduce an `sql_database` type in DataScript to model databases. `sql_table` instances may only be created as members of an `sql_database`.

```
sql_table GeoMap
{
  // see above
};

sql_database TheWorld
{
  GeoMap europe;
  GeoMap america;
};
```

## 15.4. SQL Integers

Some SQL engines internally always use an integer key or rowid. If the user-defined primary key is an integer, it can be used as row id. If the primary key is composite, it is mapped internally to an integer rowid. To avoid this indirection which requires additional storage space and increases access times, we introduce SQL integer types. An `sql_integer` is a sequence type whose members are of integer base type such that the total size of the `sql_integer` type does not exceed 64 bits. Members of an `sql_integer` may not be optional.

```
sql_integer TileId
{
  uint8  levelNr;
  int32  tileNr;
};
```

In this example, the value used as SQL key is `(levelNr << 32 + (uint32) tileNr)`.

# References

[Back] Godmar Back. *DataScript - a Specification and Scripting Language for Binary Data [http://www.cs.vt.edu/~gback/papers/gback-datascript-gpce2002.pdf]*. Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002), published as LNCS 2487. ACM. Pittsburgh, PA. October 2002. pp. 66-77.

[DataScript] *DataScript Reference Implementation* *[http://datascript.sourceforge.net]*.

[rds] *Relational DataScript [http://dstools.sourceforge.net]*.

[SQLite] *SQLite Embedded Database [http://www.sqlite.org]*.