DISPEL Language Reference

(Draft Proposal)

February 6, 2012

The Data-Intensive Systems Process Engineering Language (DISPEL) is a workflow composition language for distributed data-intensive applications. Tasks are modelled as data streams flowing through processing elements, charting workflow graphs. Graphs can be submitted to gateways; a gateway refers to a registry in order to identify suitable implementations of workflow components before delegating their execution to available resources. DISPEL is imperative and statically-typed, with strict, call-by-value evaluation of expressions.

Preamble

Language constructs are described within this reference in Extended Bachus-Naur Form. Whitespace between tokens is only necessary to distinguish tokens and is consumed on parsing. Productions enclosed within special sequences (demarked by ?) are lexical constructs which do not permit whitespace within them except where explicitly specified. Productions are ordered by precedence.

Lexical primitives  Lexical primitives used in productions are described as POSIX basic regular expressions below:

```
character = .
letter = [A-Za-z]
uppercase = [A-Z]
lowercase = [a-z]
digit = [0-9]
escape-sequence = \[tbnrf'\]
end-of-line = \[\n\]
```

Comments  Scripts can contain comments which will be ignored upon parsing. Two types of comment exist — single-line comments and multi-line comments:

```
comment = ? "//" { character - end-of-line } end-of-line ?
| ? "/*" { character - "*/" } "*/" ? ;
```

Comments are considered to be a form of whitespace for grammatical purposes.
1 Workflow model

A DISPEL workflow is an abstract network of processing elements through which data can be streamed in order to perform some data-intensive task.

- A processing element (PE) describes a persistent computational entity. Every PE has a number of connection interfaces through which data is either consumed or produced. Data is streamed between PE instances via connections made between output and input interfaces.

- A connection streams data from one output interface to at least one input interface. A connection is established by linking an output to an input and is given additional channels by linking additional inputs to the original output. Data streamed through a connection is replicated across all of its channels.

Workflows are produced by DISPEL scripts. A script is executed by submitting it to a gateway. The gateway interprets the script, exporting and importing workflow components to and from its associated registry. Interpretation results in the construction of a workflow graph which can then be used to coordinate the deployment of tasks onto a suitable enactment platform.

1.1 Workflow graphs

The workflow(s) constructed by a DISPEL script are described by an annotated directed graph constructed incrementally during interpretation. In a workflow graph, nodes have interfaces to which edges (connections) can be attached:

- PE nodes represent PE instances; each node records the type of PE used and its configuration and has a number of input and output interfaces as described by its connection signature. PE nodes are created upon instantiation of PE instances.

- Stream nodes represent data streams defined in the script itself and injected directly into a workflow; each node has a single output interface [§3.1.3].

- Sink nodes represent data-sinks; each node permits a single inbound connection. Sink nodes are created whenever a connection is asserted which goes to any of the sink types described in §1.3 below.

- External nodes represent data sources or data destinations external to the workflow; external inputs have a single output interface whilst external outputs have a single input interface. Applicable only to internal workflows within composite PEs [§4.9].

Each node in a workflow graph has a number of input and output interfaces through which to consume and produce data. Each interface can have a structural [§3.2] and a domain [§3.3] type, constraining the syntactic structure and semantic meaning respectively of data elements passed through it. A connection can only carry data which exists in the intersection of its interfaces’ structural and domain types.
There are three kinds of PE from the perspective of a DISPEL gateway. **Primitive** PEs are PEs which can be directly implemented by the gateway. **Composite** PEs are PEs which can be implemented as a composition of other PEs. **Abstract** PEs are PEs for which no implementation exists; abstract PEs cannot be instantiated until implemented as a primitive or composite PE using a constructor [§4.18].

Connections created without nodes to connect both ends are *untethered*. Sub-graphs need not be connected; isolated sub-graphs represent independent workflows and can be submitted to a gateway separately or collectively. Sub-graphs with untethered connections cannot be submitted. The workflow graph for a DISPEL script is initially empty. PE constructors produce their own independent workflow graphs.

### 1.2 Streaming

A data stream is a sequence of data elements which can be channelled in series through a connection in a workflow. Data elements can be arbitrarily complex, but must have a recognisable structural type. Data streamed through a connection can be assumed to consist of literal values and tokens. Tokens are used to describe the logical structure of streamed content:

- **SoS / EoS** denotes the start / end of the stream.
- **SoA / EoA** denotes the start / end of an array of values.
- **SoL / EoL** denotes the start / end of a list of values.
- **SoD / EoD** denotes the start / end of a dictionary of values.
- **NmD** (no-more-data) is a special token, streamed back through a connection to request the cessation of streaming.

Dictionary elements [§3.1.5] are streamed key then value. File elements are references to files, which are transferred in parallel using a file transfer protocol. Each PE instance has its own data requirements which affect consumption and production of data; buffering of data within connections is handled automatically by the enactment platform so as to ensure steady data-flow.

### 1.3 Data sinks

Data can be streamed out of a workflow via a data-sink. There are four built-in data-sink types:

- **discard** immediately disposes of any data streamed into it.
- **error** writes any data streamed into it to a locale specified by the gateway for error output.
- **warning** writes any data streamed into it to a locale specified by the gateway for warning output.
- **terminate** acts as **discard**, but immediately sends back a **NmD** token upon receiving any data.
Sinks error and warning should not be expected to be able to handle large volumes of data. Conventional results should be retrieved from a workflow by streaming them to a suitable PE instance which can perform external operations.

1.4 Submission

Upon submission of a workflow sub-graph, PE nodes will be assigned implementations in accordance with information from the registry associated with the chosen gateway. Stream nodes will be implemented by anonymous PEs which will generate the desired streams. Composite PE nodes [§3.1.6] will be replaced by their internal workflows; the connections within the internal workflow assigned to the composite PE’s interfaces will be mapped to its connections in the external graph [§4.18]. Any new nodes introduced will be assigned implementations recursively until a flattened workflow graph is created composed only of primitive components. All components of a flattened sub-graph will then be deployed for enactment regardless of graph connectivity.

1.5 Termination

A workflow terminates when all of its components terminate. A PE instance terminates when all output connections close, all input connections close or it terminates itself. When a PE terminates, the EoS (end-of-stream) token is sent through any open output connections, and a special token NmD (no-more-data) is sent back through any open input connections.

A connection is closed upon streaming an EoS token. Individual channels within a connection will be closed if a NmD token is sent back through the channel from its exit interface. If all channels are closed in this way, then the connection as a whole will close, passing back a NmD token through its entry interface. Note that a connection channel will continue to stream data until the NmD token is received and recognised.

2 Script composition

A DISPEL script is composed of a series of statements to be executed in sequence:

\[
\text{compilation-unit = statement-block \{ statement-block \} ;}
\]

Scripts are often partitioned into statement blocks delimited by braces:

\[
\text{statement-block = statement \{ "\{ \" \{ statement-block \}" \} ;}
\]

Statement blocks can be nested; a statement is considered to be within every statement block it physically resides between the delimiters of.

2.1 Namespaces

Logical entities such as variables and types are given identifiers:
The entity referred to by a given identifier is determined by the local *namespace*. A mapping between identifier and entity is inserted into the namespace whenever a statement introduces an entity and is removed at the end of the innermost statement block within which that statement resides. More recent mappings override older ones until removed. Each DISPEL script has its own namespace, as does each function, PE type and PE constructor.

Some entities have sub-components. These components can be accessed by prefacing their identifiers with a reference to their parent entities:

```
reference = { reference "," } 
( identifier | function-call ) [ "[" expression "]" ] ;
```

Entities can be referenced by their identifiers, or by a suitable function call [§4.17]. Array elements require that their index be specified [§3.1.4].

### 2.2 Registered entities

PE types, structural types, domain types, PE constructors and functions can be exported (registered) and then imported into different scripts. Registered entities are arranged hierarchically into packages. Any registered entity can be referred to without being imported into the namespace by prefacing its identifier with that of its parent package:

```
registered-entity = package-name "," identifier ;
```

Being hierarchical, package names are sequences of identifiers describing the path from a root package to a desired sub-package:

```
package-name = { identifier "," } identifier ;
```

Thus `registered-entity` is a kind of `reference`. Any node in the hierarchy can contain any number of entities and sub-packages.

### 2.3 Variable declaration

A variable is an entity which can store a value or set of values during execution of a script:

```
variable = { variable "," } ( identifier | array-element ) ;
```

Some entities have internal variables which can be referred to by prefixing their identifiers with that of the parent entity; elements of an array are accessed by affixing to the array identifier an integer expression reducing to an index [§3.1.4]. A variable declaration introduces one or more new identifiers into the namespace representing variables of a given language type [§3.1]:

```
declaration = language-type identifier [ "=" expression ]
{ "," identifier [ "=" expression ] } ;
```

Each variable can be immediately assigned the value of a corresponding expression [§2.4]. This value must be of the same type as the variable.
2.4 Expressions

An expression describes an operation which reduces upon evaluation to a value of a particular type:

\[
expression = literal-value \mid reference \mid "(" \ expression \ "")" \mid cast-expression \mid initialisation \mid assignment \mid dictionary-assignment \mid ternary-expression \mid logical-expression \mid arithmetic-expression ;
\]

Expressions apply suitable operators to literals, variable references and function calls [\S2.4.7]. Operations can be nested, their default precedence manipulated using parentheses. Expressions can also be decomposed by the language type of the value to which they reduce (e.g. integer-expression, stream-expression).

2.4.1 Casting

A cast can be performed to change the type of the value an expression reduces to by prefixing the expression with the desired type:

\[
cast-expression = "(" \ language-type \ "")" \ expression ;
\]

An Integer can be cast into a Real and vice versa (returning the floor of the real value). A String can be cast into either an Integer or a Real if an integer or real literal is embedded between the string quotes:

\[
castable-string = ? '"' \{ escape-sequence \mid " " \} [ "-" ] digit \{ digit \} \{ digit \} [ "." digit \{ digit \} ] \{ escape-sequence \mid " " \} '"' ? ;
\]

A cast from String to Integer will be the floor of any embedded real value. A reference to a PE instance can be cast into any PE type which its actual PE type is a sub-type of [\S3.1.6].

2.4.2 Initialisation

Instances of arrays and PEs require initialisation before they can be referred to:

\[
initialisation = array-initialisation \mid PE-initialisation ;
\]

Array initialisation is described in \S3.1.4, PE initialisation in \S3.1.6.

2.4.3 Assignment

A variable assignment changes the value stored within a variable by application of a suitable assignment operator [\S2.4.7]:

6
The value of any expression must be of the same type as the variable. When used within an expression, an assignment returns a value based on the assignment operator used.

2.4.4 Ternary operator

The ternary operator can be used to make conditional expressions:

\[
\text{ternary-expression} = \text{boolean-expression} \, "?" \, \text{expression} \, ":" \, \text{expression} ;
\]

Whilst the first operand must reduce to a value of type `Boolean`, the other operands can be of any same type.

2.4.5 Logic

Logical operations reduce to `Boolean` values. Comparisons can only be performed between operands of the same type whilst disjunctions, conjunctions and negations can only be performed on operands of type `Boolean` [§2.4.7]:

\[
\text{logical-expression} = \text{expression} \, ("==" \, "!=" \, "<=" \, ">=" \, ">" \, ",&&" \, "||" \, "!" \, ");
\]

PE instances are only considered equal if the two given identifiers refer to the same entity. Comparison checks are performed in accordance with the rules for each type [§3.1]. Disjunction and conjunction are non-strict; the second operand will only be evaluated if necessary to determine the value of whole expression.

2.4.6 Arithmetic

Arithmetic operations can be performed on expressions which reduce to values of type `Integer` and `Real` — or `String` and `Stream` for string and stream concatenation respectively [§2.4.7]:

\[
\text{arithmetic-expression} = \text{expression} \, ("+" \, "-" \, "*" \, "/" \, "%" \, "+=" \, "-=" \, "+=" \, "-=" \, ");
\]

Both operands in any binary operation must reduce to the same type.

2.4.7 Operators

All valid assignment operators are described in Table 1 in order of descending precedence. Likewise, all valid expression operators are described in Table 2.
var ++ Increments the value assigned to var by one; returns the unincremented value (Integer and Real variables only).

var -- Decrements the value assigned to var by one; returns the undecremented value (Integer and Real variables only).

++ var Increments the value assigned to var by one; returns the incremented value (Integer and Real variables only).

-- var Decrements the value assigned to var by one; returns the decremented value (Integer and Real variables only).

var = exp Assigns the value of exp to var; returns that value (both operands must be of the same type).

var += exp Appends the value of exp to that of var; returns the concatenation (both operands must be Strings or both Streams).

var += exp Adds the value of exp to that of var; returns the sum (both operands must be Integers or both Reals).

var -= exp Subtracts the value of exp from that of var; returns the difference (both operands must be Integers or both Reals).

var *= exp Multiplies the value of var by that of exp; returns the product (both operands must be Integers or both Reals).

var /= exp Divides the value of var by that of exp; returns the quotient (both operands must be Integers or both Reals).

Table 1: Assignment operators

2.5 Annotations

Annotations are used to provide additional information about a given entity:

\[
\text{annotations} = \text{annotation} \{ \",\" \text{annotation} \} ;
\]

An annotation is a key-value pair where the value must reduce to a string literal and the key is a special identifier preceded by an @ symbol:

\[
\text{annotation} = \text{descriptor} \"=\" \text{expression} ;
\]

\[
\text{descriptor} \equiv \? \"@" \{ \text{letter} | \text{digit} | \"\_\" \} ?
\]

Annotations wield no influence in DISPEL, but may be used to pass information to the enactment platform or as an additional mode of documentation. Annotations can be attached to any registrable entity [§4.4] during specification or registration and to PE instances [§3.1.6]; all annotations are recorded upon registration regardless of provenance.

3 Type System

DISPEL has three type systems: language types refer to the types of variables in scripts; structural types refer to the syntactic structure of data elements streamed between PE instances; domain types refer to the semantic (principally ontological) meaning assigned to data elements.
++ exp \quad \text{Increments the value of exp by one; if exp is a variable, then the value assigned to exp will be incremented by one (Integer and Real only).}

-- exp \quad \text{Decrements the value of exp by one; if exp is a variable, then the value assigned to exp will be decremented by one (Integer and Real only).}

- exp \quad \text{Negates the given expression (Integer and Real only).}

! exp \quad \text{Evaluates the logical negation of the operand (Boolean only).}

exp * exp \quad \text{Multiplies both operands together (both operands must be Integers or both Reals).}

exp / exp \quad \text{Divides the left operand by the right (both operands must be Integers or both Reals).}

exp \% exp \quad \text{Returns the remainder when the left operand is divided by the right (both operands must be Integers or both Reals).}

exp + exp \quad \text{Concatenates two expressions such that the left operand precedes the right (both operands must be Strings or both Streams).}

exp + exp \quad \text{Sums the values of two expressions (both operands must be Integers or both Reals).}

exp - exp \quad \text{Subtracts the right operand from the left (both operands must be Integers or both Reals).}

exp < exp \quad \text{Evaluates whether or not the left operand is less than the right operand (comparables only).}

exp > exp \quad \text{Evaluates whether or not the left operand is greater than the right operand (comparables only).}

exp <= exp \quad \text{Evaluates whether or not the left operand is less than or equal to the right operand (comparables only).}

exp >= exp \quad \text{Evaluates whether or not the left operand is greater than or equal to the right operand (comparables only).}

exp == exp \quad \text{Equality check.}

exp != exp \quad \text{Inequality check.}

exp && exp \quad \text{Evaluates the logical conjunction of both operands (boolean only).}

exp || exp \quad \text{Evaluates the logical disjunction of both operands (boolean only).}

exp ? exp : exp \quad \text{Returns the middle operand if the left operand is true; otherwise, the right operand is returned.}

Table 2: Expression operators

3.1 Language types

Each variable has a language type which restricts the set of values which can be attributed to it:

\text{language-type = primitive-type | reference-type ;}

Variables of primitive types store literal values:

\text{primitive-type = "Boolean" | "Integer" | "Real" | "String" ;}

Variables of reference types store references to objects. These references are internal values which can be copied from one variable to another without replicating the object being referred to:
References cannot be directly manipulated. language-identifier is a kind of identifier for constructed types such as PEs defined using Type declarations [§4.5] or imported from a registry [§4.3].

3.1.1 Primitive types

An expression of a primitive type will reduce to literal value which can be stored within any variable of that type. Each primitive type has its own set of literals:

```
literal-value = boolean-literal | integer-literal | real-literal | string-literal
```

A Boolean can be true or false:

```
boolean-literal = "true" | "false";
```

An Integer can be any integer number:

```
integer-literal = ["-" ]? digit { digit }?
```

A Real can be any decimal number:

```
real-literal = ["-" ]? digit { digit }[ "." digit { digit } ]?
```

A String can be any sequence of characters:

```
string-literal = ?"" { escape-sequence | character - ("" | ")" ) }""?
```

Strings can contain escape characters which represent non-printable elements such as carriage returns and tabs, as well as double-quotes themselves.

3.1.2 Connections

A Connection is a reference to a connection. A value can only be assigned to a Connection variable by instantiating a PE which it is an interface of [§3.1.6], passing it to a connection declaration [§4.8] or assigning another Connection variable to it. An interface of a PE instance cannot be assigned another Connection variable.

3.1.3 Streams

A Stream is a reference to either a sequence of primitive literals and stream tokens, or a stream comprehension describing such a sequence. A sequence can be constructed by assigning a Stream variable a concatenation of streamable expressions and stream tokens [§1.2], or a stream literal:

```
stream-literal = "|-" [ streamable-expression
  { "," streamable-expression } ] "-|
```

A streamable-expression is a kind of expression which reduces to a primitive value or to a reference to an instance of a streamable constructed type. A constructed type instance is streamable if it contains only primitive values or references to streamable constructed type instances. Alternatively, a Stream variable can be assigned a stream comprehension:

```
stream-comprehension
  = "|-" "repeat" ( "enough" | integer-expression ) "of"
    ( streamable-expression | "{" streamable-expression }
      \"\"
    } "|-" ;
```

A comprehension repeat expression of sequence describes a recurring sequence. If streamed through a connection [§4.8], sequence should repeat expression times, or if expression is enough, until a NmD token is received by the source. Expressions within comprehensions are evaluated upon execution as normal; comprehensions themselves are computed incrementally when streamed.

### 3.1.4 Arrays

An array variable stores a reference to an array. An array is an ordered list of values of the same type, indexed numerically by offset. An array is constructed by affixing \[] to the type of its elements:

```
array-type = language-type \[ \"] ;
```

A multi-dimensional array can be created by constructing an array of arrays. An array must be initialised before it can be referred to by assigning an array literal or specifying the array size:

```
array-initialisation = "{\[
  expression \\{}
  "{" expression \}\"
\"
  | "new" language-type \[ integer-expression \]
  \{
    \"\integer-expression \"] \}
 ;
```

An array literal is an ordered sequence of expressions of the expected type equal in length to the array to which it is assigned; the value of each expression is assigned to the corresponding array element. A multi-dimensional array must be initialised in a single statement by specifying the array size of each nested array in sequence.

Once an array has been initialised, values can be freely assigned to its elements. An array element is referred to by affixing to a reference to the array an expression which reduces to the index of the desired element:

```
array-element = reference \[ expression \] ;
```

Thus array-element is itself a kind of reference. Indices start at zero and increment by one for each element after the first.

### 3.1.5 Dictionaries

A dictionary variable stores a reference to a dictionary, an unordered composition of labelled values of different language-types. A dictionary is constructed by declaring its constituent elements within <> brackets:
dictionary-type
    = "<" [ language-type-group { ";" language-type-group } ] "">" ;

language-type-group = language-type identifier { "," identifier } ;

Specification of elements of the same type can be grouped together. Each element has its own identifier, which must be unique within the dictionary. Elements can be assigned values individually, or collectively:

dictionary-assignment = "<" [ identifier "=" expression { "," identifier "=" expression } ] "">" ;

A dictionary assignment is a mapping of expressions to specific entities in the dictionary; not all entities need be assigned values in a single assignment. Individual elements in a dictionary are accessed by affixing their identifiers to a reference to the dictionary:

dictionary-element = reference "." identifier ;

Thus dictionary-element is itself a kind of reference. A dictionary without elements is denoted by the empty dictionary <>.

3.1.6 Processing elements

A PE specification is considered within DISPEL to be a constructed type. A new PE type can be constructed using an Type declaration §4.5 where:

PE-type-constructor
    = ( "PE" "(" { stype-declaration ";" } { dtype-declaration ";" } connection-signature ")" | PE-type )
        [ "with" ( configuration { "," annotations } | annotations ) ] ;

A PE type is described by its connection signature or by adapting an existing PE type; thus PE-type is a kind of language-identifier. A connection signature describes a PE’s connection interfaces as a dictionary of inputs linked to a dictionary of outputs:

connection-signature = interfaces "=>" interfaces ;

interfaces = "<" [ interface-group { ";" interface-group } ] "">" ;

Each input and output connection interface is an augmented Connection or Connection array declaration with the optional addition of a structural type §3.2, a domain type §3.3 and any number of connection modifiers:

interface-group
    = "Connection" [ ":[ " ] ]
      [ ":" structural-type ] [ ":=" domain-type ]
      { modifier } identifier { "," identifier } ;

Connection modifiers are used to modify the behaviour of connection interfaces.

modifier = modifier-id [ ":" expression { "," expression } ""] ;

Some modifiers take parameters — these parameters are listed in parentheses immediately after the modifier. All valid modifier-ids are described in Table 3.
after Used to delay the consumption of data through one or more connections. [Requires a list of predecessors.]
compressed Used to compress data streamed out of the modified connection or to identify the compression used on data being consumed when applied to an output or an input interface respectively. [Requires a compression scheme.]
default Used to specify the default input streamed through a connection should input be otherwise left unspecified. [Requires a stream expression; input only.]
encrypted Used to encrypt data streamed out of the modified connection or to identify the encryption scheme used on data being consumed when applied to an output or an input interface respectively. [Requires an encryption scheme.]
initiator Used to identify connections which provide only an initial input before terminating. Inputs marked initiator are read to completion before reading from inputs not so marked. [Input only.]
limit Used to specify the maximum number of data elements a connection will consume or produce before terminating. [Requires a positive integer value.]
locator Used where the modified connection indicates the location of a resource to be accessed by the associated PEI (which might influence the distribution of the workflow upon execution). [Input only.]
lockstep Indicates that one data element must be streamed through every interface in the modified array before another element can be streamed through any of them. [Connection arrays only.]
permutable Indicates that a given array of inputs can be read from in any order without influencing the outputs of the PEI. [Input connection arrays only.]
preserved Indicates that data streamed through the modified connection should be recorded in a given location. [Requires a URI, or goes to a default location.]
requiresDtype Dictates that upon instantiation, the specific domain type of the modified connection must be defined.
requiresStype Dictates that upon instantiation, the specific structural type of the modified connection must be defined.
roundrobin Indicates that a data element must be streamed through each interface in the modified array in order, one element at a time. [Connection arrays only.]
successive Indicates that each interface of the modified array must terminate before the next one is read. [Connection arrays only.]
terminator Causes a PEI to terminate upon the termination of the modified connection alone (rather than once all inputs or all outputs have terminated).

Table 3: Connection modifiers

If an interface omits an explicit structural type, then it is of type Any; if it omits an explicit domain type, then it is of type Thing. Interfaces with the same structural type, domain type and connection modifiers can be grouped together as normal for dictionary elements. An instance of a PE must be created before it can be connected to a workflow:
PE-initialisation = "new" language-type [ "with" ( configuration {"," annotations } | annotations ) ] ;

configuration = configurable { "," configurable } ;

The length of a Connection array can be defined for PE instances. Connection interfaces can also be redefined, adding new connection modifiers or restricting their structural and domain types:

configurable = [ instance "." ] "length" "=" expression
| [ identifier "as" ] [ interface-type ] { modifier } identifier ;

Only existing interfaces can be redefined in this way; interfaces can be renamed however by declaring old as... new, where old is the original interface identifier and new is its replacement. A PE’s interfaces are identified within the local namespace for the duration of the statement, allowing interfaces to be referred to without prefix.

A PE variable may only store a reference to an instance of a PE which is a sub-type of the variable’s type. A PE Child is a sub-type of PE Parent if and only if any instance of Child can replace an instance of Parent in any workflow:

- Each input interface of Parent must have a structural and domain type which are sub-types of those of an input interface of Child with the same name [§3.2.5, 3.3.3]. If Child has any additional input interfaces, then they must each have a default property.
- For each output interface of Parent, Child must have an output interface of the same name with structural and domain types which are sub-types of those of Parent’s interface.

### 3.2 Structural types

Every connection has a structural type which restricts the set of values which can be streamed through it:

structural-type = "Any" | simple-stype | compound-stype ;

Interfaces of type Any can stream any data value. Interfaces of simple types stream unstructured literal values:

simple-stype = "Boolean" | "Integer" | "Real"
| "String" | "Byte" | "File" ;

Interfaces of compound types stream structured data, which is composed of data with their own structural types:

compound-stype = array-stype | list-stype | dictionary-stype
| constructed-type ;

Aliases for compound types are defined using Stype declarations [§4.6] or imported from a registry [§4.3].
3.2.1 Simple types

Any data element can be decomposed into one of six data types.

- A **Boolean** element can be true or false.
- An **Integer** element is any whole number.
- A **Real** element is any number.
- A **String** element is any fragment of text.
- A **Byte** element is a single byte of binary data.
- A **File** element is a reference to a discrete file.

DISPEL does not assume a particular representation of primitive structural types — this is handled by the gateway. The File type is treated logically as if embedded in a data stream; in practice only a reference to a file is streamed whilst a parallel file transfer is facilitated by the enactment platform.

3.2.2 Arrays

An array is an ordered list of values of the same structural type. An array is described by affixing `[]` to the type of its elements:

    array-stype = structural-type "[" "]" ;

A multi-dimensional array can be created by describing an array of arrays. An array is streamed by first sending the **SoA** token followed by size of the array; each element of the array is then streamed in turn, finishing with the **EoA** token. Note that the size of an array is not specified with a script, but is determined during execution of a workflow.

3.2.3 Lists

A list is an array of unspecified size. A list is described by embedding the type of its elements within `[]` brackets:

    list-stype = "[" structural-type "]" ;

A list is streamed by first sending the **SoL** and then sending each element of the list in turn, finishing with the **EoL** token.

3.2.4 Dictionaries

A dictionary is an unordered set of labelled values of different structural types. A dictionary is described by declaring its constituent elements within `<>` brackets:

    dictionary-stype = "<" [ "rest" | structural-type-group
    
    { ";" structural-type-group } [ ";" "rest" ] ] ">" ;

    structural-type-group = structural-type identifier { "," identifier } ;
Specification of elements of the same type can be grouped together. If an interface is only concerned with some of the values in a dictionary, then only the identifiers for those values of interest need be specified, followed by the `rest` keyword. `rest` must be the final component of a dictionary and represents zero or more unidentified data elements. A dictionary is streamed by first sending the `SoD` token and then sending each element of the dictionary in turn, label then value, finishing with the `EoD` token.

### 3.2.5 Sub-typing

Whether or not a structural type \( ST' \) is a sub-type of another structural type \( ST \) \( (ST' \subseteq ST) \) is determined by the following rules:

- \( \forall ST. ST \subseteq Any \).
- \( \forall ST. ST \subseteq ST \).
- \( [ST'] \subseteq [ST] \) if and only if \( ST' \subseteq ST \).
- \( ST'[] \subseteq ST[] \) if and only if \( ST' \subseteq ST \).
- \( < ST'_1; \ldots; ST'_m; rest > \subseteq < ST_1; \ldots; ST_n; rest > \) if and only if \( m = n \) and, after sorting identifiers according to a standard scheme, \( id'_i = id_i \) and \( ST'_i \subseteq ST_i \) for all \( i \) such that \( 1 \leq i \leq n \).
- \( < ST'_1; \ldots; ST'_m; rest > \subseteq < ST_1; \ldots; ST_n; rest > \) if and only if \( m \geq n \) and there exists a permutation of identifiers \( id'_1, \ldots, id'_m \) such that \( id'_i = id_i \) and \( ST'_i \subseteq ST_i \) for all \( i \) such that \( 1 \leq i \leq n \).

### 3.3 Domain types

Every connection interface has a domain type which ascribes a semantic interpretation to the data streamed through it:

\[
\text{domain-type} = \text{"Thing" | domain-identifier | domain-descriptor} \\
\quad | \text{domain-array | domain-list | domain-dictionary};
\]

`Thing` is the super-type of all domain types, and is implied where no domain type is specified. A `domain-identifier` is an identifier representing a domain descriptor or constructed domain type \( \S 4.7 \).

#### 3.3.1 Domain descriptors

A domain descriptor is a direct reference to an element in an ontology:

\[
\text{domain-descriptor} = \\
\quad ? "" \text{character - ( "" | \"\" ) \{ character - ( "" | \"\" ) \} } \\
\quad "" : \text{character - ( "" | \"\" ) \{ character - ( "" | \"\" ) \} } "" ;
\]

A domain descriptor is a domain namespace identifier \( \S 4.2 \) prefixing an ontology element name. Domain descriptors can be substituted for domain type identifiers when specifying the domain type of a connection interface \( \S 3.1.6 \).
3.3.2 Compound domain types

Compound domain types can be described in the same form as for structural types:

```
domain-array = domain-type "[" "]";

domain-list = "[" domain-type "]";

domain-dictionary = "<" [ "rest" | domain-group { ";" domain-group } [ ";" "rest" ] ] ">";

domain-group = domain-type identifier { "," identifier };
```

The structure of a domain type must correspond to the structure of the structural type assigned to the same interface; it is permissible however for a simple domain type to correspond to a compound structural type.

3.3.3 Sub-typing

Whether or not a domain type $\text{DT}'$ is a sub-type of another domain type $\text{DT}$ ($\text{DT}' \sqsubseteq \text{DT}$) is determined by the following rules:

- $\forall \text{DT}. \text{DT} \sqsubseteq \text{Thing}$.
- $\forall \text{DT}. \text{DT} \sqsubseteq \text{DT}$.
- $\text{DT}' \sqsubseteq \text{DT}$ if $\text{DT}'$ represents an ontology element "o:x", $\text{DT}$ represents an ontology element "o:y" and $x$ is a sub-type of $y$ according to ontology o.
- $[\text{DT}]' \sqsubseteq [\text{DT}]$ if and only if $\text{DT}' \sqsubseteq \text{DT}$.
- $[\text{DT}]' \sqsubseteq [\text{DT}]$ if and only if $\text{DT}' \sqsubseteq \text{DT}$.
- $< \text{DT}_1, \text{id}'_1; \ldots; \text{DT}_m, \text{id}'_m > \sqsubseteq < \text{DT}_1, \text{id}_1; \ldots; \text{DT}_n, \text{id}_n >$ if and only if $m = n$ and, after sorting identifiers according to a standard scheme, $\text{id}'_i = \text{id}_i$ and $\text{DT}'_i \sqsubseteq \text{DT}_i$ for all $i$ such that $1 \leq i \leq n$.
- $< \text{DT}_1, \text{id}'_1; \ldots; \text{DT}_m, \text{id}'_m > \sqsubseteq < \text{DT}_1, \text{id}_1; \ldots; \text{DT}_n, \text{id}_n; \text{rest} >$ if and only if $m \geq n$ and there exists a permutation of identifiers $\text{id}_1', \ldots, \text{id}_m'$ such that $\text{id}_i' = \text{id}_i$ and $\text{DT}_i' \sqsubseteq \text{DT}_i$ for all $i$ such that $1 \leq i \leq n$.

4 Statements

A DISPEL script is built from simple and compound statements executed in series:

```
statement = package-statement | namespace-statement | import-statement
      | registration | variable-declaration ";" |
      | ltype-declaration | stype-declaration | dtype-declaration
      | connection-declaration | composition | if-statement
      | switch-statement | while-statement | for-statement
      | break-statement | continue-statement | function
```
Compound statements include statement blocks which are themselves executed in series during execution of the overall statement.

4.1 package statement

A statement block can be associated with a particular package, automatically adding entities registered within that package to the namespace for the extent of that statement block:

```plaintext
package-statement = "package" package-name statement-block ;
```

`package` associates the enclosed statement block with the given package then executes it. If the `package` statement is already associated with a package, then the association is overridden within the enclosed statement block. The package referred to need not exist prior to execution of the statement.

4.2 namespace statement

Identifiers can be conferred to outside ontologies, allowing references to elements within those ontologies:

```plaintext
namespace-statement = "namespace" identifier string-literal ";" ;
```

`namespace` maps an identifier to the given ontology URI in the local namespace, permitting the use of domain descriptors prefixed with that identifier [§3.3.1].

4.3 import statement

A `use` statement imports the given registered entities into the namespace, obviating the need to preface their identifiers with those of their parent package:

```plaintext
import-statement = "use" package-name "." 
    ( identifier | "{" identifier { "," identifier } "}" ) ";"
```

Multiple entities from the same package can be imported simultaneously. Entities within package `dispel.lang` are imported into the namespace automatically at the beginning of every script for its full extent.

4.4 register statement

A `register` statement registers one or more entities:

```plaintext
registration = "register" identifier { "," identifier } 
    [ "with" annotations ] ";" ;
```

Each entity will be registered within the package associated with the `register` statement with the annotations provided. If no such package exists, then `register` cannot be invoked. If an entity with the same identifier already exists
within the package associated with the register statement, then that entity will be overridden.

If a function §4.17 or PE constructor §4.18 is registered, then its namespace information will also be recorded so as to permit the import of entities upon which it depends. If any such entities are unregistered, then they will be registered automatically prior to registration of the dependent entity with no additional configuration.

### 4.5 Type declaration

New constructed language types can be defined using Type declarations:

```
ltype-declaration
  = "Type" language-identifier "is" ( language-type "with" annotations | PE-type-constructor ) ";" ;
```

A constructed type can simply be an alias for an existing language type (usually a compound type like a dictionary), or it can describe a new abstract PE type §3.1.6. The identifier for the new type will be inserted into the local namespace.

### 4.6 Stype declaration

An stype declaration allows the creation of aliases for compound structural types:

```
stype-declaration = "Stype" structural-identifier "is" structural-type "with" annotations ";" ;
```

The identifier for the new type is added to the local namespace. If declared within a PE type specification §3.1.6 and attributed to any interface of that PE, then any unification of the declared type with another type in a connection §4.8 will result in a commensurate unification with all instances of the declared type used in other interfaces of that PE instance.

### 4.7 Dtype declaration

A Dtype declaration derives a new domain type from an existing type:

```
dtype-declaration
  = "Dtype" domain-identifier ( "is" domain-type [ "represents" domain-descriptor ] | 
                             "represents" domain-descriptor ) "with" annotations ";" ;
```

The identifier for the new type is added to the local namespace. A constructed domain type can be directly associated with an element in an ontology — any compound type constructed using the new domain type will be assumed to be a compound involving the represented element, and likewise any sub-type relation with the domain type will be assumed to describe an equivalent relationship with the element. As with structural types, declaring a domain type within a
PE type specification will create sympathy between interfaces attributed that domain type [§4.6].

4.8 Connections

A connection declaration forms a new connection or adds a channel to an existing connection:

\[
\text{connection-declaration} = ( \text{reference} \mid \text{stream-expression} ) \\
\text{"=>" ( reference } \mid \text{data-sink}) ;
\]

By default, a declaration \text{entry => exit} merges two connections in the local workflow graph referred to by two Connection variables \text{entry} and \text{exit}. Two connections can merge provided that at least one connection is untethered at its source; the new connection will inherit the union of both parents’ channels, the intersection of both parents’ structural and domain types [§3.2.5 and §3.3.3] and be referred to by both \text{entry} and \text{exit}. Alternatively, \text{entry} can be a Stream, in which case a connection emitting from a stream node will be created and merged with \text{exit}. Finally, \text{exit} can be a data-sink [§1.3]:

\[
\text{data-sink} = \text{"discard" } \mid \text{"error" } \mid \text{"warning" } \mid \text{"terminate" ;}
\]

A connection directed to a sink node will be created and merged with \text{entry}.

4.9 Processing element composition

A PE composition statement creates a new composite PE by applying a PE constructor [§4.18] to an abstract PE:

\[
\text{composition} = \text{"PE" } \langle \text{PE-type } \rangle \text{ PE-type } \text{"=" constructor-call } ;
\]

A statement \text{PE<AbstractPE> CompositePE = constructor(...)} will apply constructor \text{constructor} to abstract PE \text{AbstractPE}, creating a new (composite) PE type \text{CompositePE} which can be instantiated. The connection signature of \text{AbstractPE} must match that used by \text{constructor}.

4.10 if conditional

An if statement only permits the execution of a statement block if a given boolean expression evaluates as true:

\[
\text{if-statement} = \text{"if" } \langle \text{" boolean-expression "} \rangle \text{ statement-block}
\]

An optional else clause can provide an alternative statement block to execute if the given expression is false; otherwise, execution will skip to the next unenclosed statement.
4.11 switch conditional

A switch statement partitions a statement block — different partitions are executed based on the evaluation of a given expression:

```
switch-statement = "switch" primitive-expression "{" 
    { "case" literal-value ":" { statement } } 
    [ "default" ":" { statement } ] "}" ;
```

Partitions are marked by case markers, each associated with a literal denoting a particular value of the given expression; execution will proceed from the first matching marker until the end of the statement block unless redirected. If no matching case is found, then execution will proceed from the default marker should one exist or immediately after the statement block otherwise.

4.12 while iterator

A while iterator repeats the execution of a given statement block whilst a given boolean expression evaluates as true:

```
while-statement = "while" "(" boolean-expression ")" statement-block ;
```

If the expression becomes (or is initially) false, execution resumes immediately after the iterator.

4.13 for iterator

A for iterator describes a repetition based on a regular update of a control variable:

```
for-statement 
    = "for" "(" variable-declaration ";" boolean-expression ";;" assignment ")" statement-block ;
```

Prior to attempting to execute a statement block, the given variable declaration is executed. The statement block is executed if the given boolean expression is true and is re-executed until it becomes false, at which point execution resumes immediately after the iterator. After each iteration (if any), the given assignment is executed.

4.14 break statement

A break statement forces execution of an iterator or switch statement to end:

```
break-statement = "break" ";" ;
```

Execution is resumed immediately after the innermost iterator or switch statement in which break is located should one exist.
4.15 **continue statement**

A `continue` statement forces execution of an iteration to end:

```
continue-statement = "continue" ;
```

`continue` can only be invoked within an iterator. Execution skips ahead to the next iteration of the innermost iterator in which `continue` is located provided that the iteration condition holds; otherwise execution resumes immediately after the iterator.

4.16 **submit statement**

A `submit` statement submits (part of) the local workflow graph for enactment:

```
submission = "submit" [ instance { "," instance } ] ;
```

`submit` submits all nodes of the local workflow graph connected to at least one of the PE instances provided, or the entire graph if no instance is provided. Submission cannot be performed within a PE constructor; submission of nodes with untethered connections will fail.

4.17 **Functions**

A function is a parameterised statement block describing a recurring execution pattern:

```
function = ( "void" | language-type ) identifier parameter-tuple
            statement-block ;
```

A function is not executed immediately, but is instead invoked on demand as often as required. A function consists of a return type, an identifier (inserted into the local namespace), a set of parameters and a statement block. The return type must be a valid language type, or `void`. A tuple of variable declarations without assignments form the function parameters:

```
parameter-tuple = "(" [ language-type identifier
                        { "," language-type identifier } ] ")" ;
```

Each function has its own namespace, consisting of the local namespace at the end of the statement block in which the function is defined, extended with its parameter set.

A function is invoked using its identifier and a tuple of values; these values correspond and are assigned to the function’s parameters for the duration of the invocation:

```
function-call = identifier "(" [ expression { "," expression } ] ")" ;
```

The function is executed within its own namespace. A function can be invoked as a statement or as part of an expression unless `void`: if invoked as a statement, then execution will continue from immediately after the statement upon completion; If invoked within an expression, the function call will reduce to the
value returned by the function and execution of the expression will resume upon completion.

If a function is not `void`, then it must return a value of its return type, which a call to the function will then reduce to within any expression. This is ensured by including a `return` statement within the function:

```plaintext
return-statement = "return" [ expression ] ";";
```

A `return` statement of this kind may only be included within a function. It immediately ends execution of a function, returning the value which the provided expression reduces to should one be provided — which must be the case if the function is not `void`. If the function is not `void`, a `return` statement must be executed within a function before the end of the function.

### 4.18 Processing element construction

A PE constructor is a parameterised statement block used to specify an internal workflow graph with which to implement instances of composite PEs.

```plaintext
PE-constructor = "PE" "<" PE-type ">" identifier parameter-tuple statement-block ;
```

A PE constructor consists of a reference to an abstract PE, identifier (inserted into the local namespace), parameter tuple and statement block. It has its own namespace, consisting of the local namespace at the end of the statement block in which constructor is defined, extended with its parameter set. The constructor’s own statement block is not executed immediately, but upon every act of PE composition which uses the constructor [§4.9].

A PE constructor is invoked using its identifier and a tuple of values; these values correspond and are assigned to the constructor’s parameters for the duration of the invocation.

```plaintext
constructor-call = PE-type "(" [ expression { "," expression } ] ")" ;
```

The constructor is executed within its namespace, producing an internal workflow graph distinct from the local graph at point of invocation. A PE constructor must invoke a special `return` statement before it terminates:

```plaintext
PE-return = "return" "PE" "(" "<" interface-assignments ">" "=>"
            "<" interface-assignments ">" ")" ";"
```

```plaintext
interface-assignments = [ interface "=" connection
                         { "," interface "=" connection } ] ;
```

Every interface of the constructor’s abstract PE type must be assigned a connection in the internal workflow such that no untethered connections remain.