



ADMIRE D1.7 – DISPEL : Grammar and Concrete Syntax, version 1.0

Project Title	ADMIRE
Document Title	DISPEL : Grammar and Concrete Syntax, version 1.0
Deliverable Number	D1.7
Authorship	Malcolm Atkinson, Peter Brezany, Amy Krause Jano van Hemert, Ivan Janciak, Gagarine Yaikhom
Document Filename	ADMIRE-D1.7-research-prototypes.tex
Document Version	1.0
Distribution Classification	Project Internal
Distribution List	ADMIRE Project Team
Approval List	Jano van Hemert, Project Manager, Executive Board

<i>Personnel</i>	<i>Date</i>	<i>Comment</i>	<i>Version</i>
IJ	01 Feb 2010	First draft	0.1
GY	05 Feb 2010	DISPEL grammar	0.2
GY	05 Feb 2010	DISPEL syntax	0.3
IJ	12 Feb 2010	Executive summary/conclusions	0.4
JVH	22 Feb 2010	Reviewed	0.5
RMB	26 Feb 2010	Final edit and signoff	1.0

Contents

Executive Summary	2
1 DISPEL Context-free Grammar	5
1.1 Top-down parsing	5
1.2 Notational conventions	5
2 DISPEL Syntax	5
2.1 White spaces	5
2.2 Comments	6
2.3 Reserved words	6
2.4 Keywords	7
2.5 Punctuators	7
2.6 Identifiers	7
2.7 Literals	8
2.7.1 Null Literals	9
2.7.2 Boolean Literals	9
2.7.3 String Literals	9
2.7.4 Character Literals	9
2.7.5 Numeric Literals	9
2.7.6 Escape Sequences	10
2.8 Compilation unit	10
2.9 Import declarations	11
2.10 Statement blocks	11
2.11 Statements	11
2.11.1 Variable declaration	12
2.11.2 Variable initialisation	12
2.11.3 Connection declaration	12
2.11.4 Create connection	12
2.11.5 Conditional construct	13
2.11.6 Loop constructs	13
2.11.7 Jump statement	14
2.11.8 Exception handling	14
2.11.9 Assignment statement	14
2.12 The type system	14
2.12.1 Primitive data type	15
2.13 Expression	15
2.14 Functions	17
2.14.1 Calling a function	17
2.14.2 Creating a function	17

2.15	Creator	18
2.16	Sequences	18
2.17	Tuples	19
2.18	Input and output interfaces	19
2.19	Processing elements	19
2.19.1	Declaring a processing element	19
2.19.2	Creating a processing element instance	19
3	Conclusion and Future Work	20

Executive Summary

This document provides a specification of the Data-Intensive Systems Process Engineering Language (DISPEL) for expressing hierarchical data-flow abstractions for use in distributed data mining and integration applications. This specification defines the context-free grammar and concrete syntax of the language. The concrete textual syntax of the language supports design and interchange of the modelled data mining and integration processes.

In our approach, the concrete syntax is produced as a mapping of some of the abstract elements defined in the DISPEL abstract syntax to their concrete textual representations. The concrete textual syntax is beneficial for many reasons. It supports usability of the language and productivity of users because of its fast editing style, its usage of error markers, autocompletion, quick fixes, and many others. Furthermore, it can easily be integrated to existing tools and modified by conventional text editors or specialized graphical editors such as the ADMIRE Process Designer.

This document describes the first complete version of the DISPEL concrete syntax as used in the current version of the ADMIRE Gateway and integrated ADMIRE Workbench tools. It is meant to be used as a reference document for DMI experts as well as for developers working on the DISPEL parsers, language and workflow optimisers, DMI workbenches and DMI editors.

1 DISPEL Context-free Grammar

This section describes the context-free grammars used in this specification to define the lexical and syntactic rules of DISPEL.

1.1 Top-down parsing

We use an `LL(*)`¹ grammar, which is a top-down parser that parses from left-to-right and constructs a left-most derivation of the input tokens by looking ahead k tokens, where the value of k is adjusted dynamically during parsing. We chose this grammar so that it can be implemented immediately using the `Antlr`² parser generator, version 3.

1.2 Notational conventions

Non-terminal symbols in the lexical and syntactic rules are given in typescript font. Terminal symbols in these rules are enclosed inside single-quotes. Terminal symbols should appear in the DISPEL script exactly as written. For instance, in the following production rule, `import-declaration` and `qualified-name` are non-terminal symbols; whereas, `'use'`, `'.'`, `'*'`, and `';'` are terminal symbols. Please note that UNICODE terminal characters are denoted as `'\uXXXX'`, where `X` is a hexadecimal digit (e.g., `'\u000C'`).

```
import-declaration
: 'use' qualified-name ('.' '*')? ';'
;
```

The remaining symbols, are used to write the production rule (see table 1).

2 DISPEL Syntax

2.1 White spaces

White spaces are lexical components that are discarded before parsing, except when they appear in string literals, see: `string-literals`. They may be used for improving readability of the DISPEL source code.

```
white-space : ( ' ' | '\r' | '\t' | '\u000C' | '\n' ) ;
```

¹`LL(*)` grammars are `LL(k)` grammars where the value of k may be altered during parsing. Such grammars are supported by `Antlr`.

²www.antlr.org: ANother Tool for Language Recognition

	Description
;	Terminates a production rule.
:	Marks new production. The expression on the right defines the non-terminal on the left.
(Marks the beginning of an expression block.
)	Marks the end of an expression block.
?	Zero or one occurrence of the expression on the left.
+	One or many occurrence of the expression on the left.
*	Zero or many occurrence of the expression on the left.
	Separates alternative production rules for a given non-terminal.
~	Does not contain the following expression.
..	Value range (e.g., 1..4 means 1, 2, 3, and 4).

Table 1: Punctuation marks for defining production rules

2.2 Comments

DISPEL allows two types of comments: single-line comments or block comments, which span multiple lines. When block comments are used, the first occurrence on ‘*/’ marks the end of the block comment, i.e., the matching is not greedy.

```
comment
  : single-line-comment
  | block-comment
  ;

block-comment
  : '/*' ( . )* '*/'
  ;

single-line-comment
  : '//' ~('\n'|\r)* '\r'? '\n'
  ;
```

2.3 Reserved words

DISPEL reserves several words to assigned special meanings. These words should not be used as identifiers.

```
reserved-word
  : keyword
  | null-literal
  | boolean-literal
  ;
```

2.4 Keywords

All of the keywords reserved by DISPEL are listed in table 2. These keywords are case-sensitive (i.e., `Connection` is a keyword; whereas `connection` is not a keyword).

<code>use</code>	<code>if</code>	<code>for</code>	<code>while</code>
<code>do</code>	<code>switch</code>	<code>return</code>	<code>throw</code>
<code>break</code>	<code>continue</code>	<code>Boolean</code>	<code>Integer</code>
<code>Real</code>	<code>String</code>	<code>Identifier</code>	<code>new</code>
<code>function</code>	<code>Connection</code>	<code>PE</code>	<code>PEI</code>
<code>void</code>	<code>repeat</code>	<code>enough</code>	<code>of</code>
<code>register</code>	<code>withdraw</code>	<code>any</code>	<code>rest</code>

Table 2: DISPEL keywords

2.5 Punctuators

DISPEL uses punctuators to either structure the statements in the script, or to express operations.

<code>{</code>	<code>}</code>	<code>(</code>	<code>)</code>
<code>[</code>	<code>]</code>	<code><</code>	<code>></code>
<code>=</code>	<code>,</code>	<code>;</code>	<code>=></code>
<code>...</code>	<code>++</code>	<code>-</code>	<code>+</code>
<code>-</code>	<code>/</code>	<code>%</code>	<code>*</code>
<code>&&</code>	<code> </code>	<code>!=</code>	<code>+=</code>
<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>
<code>^=</code>	<code>&=</code>	<code> =</code>	

Table 3: DISPEL punctuators

2.6 Identifiers

```

identifier
  : Letter ( Letter | JavaIDDigit )*
  ;

```

```

Letter
  : '\u0024'
  | '\u0041'..' \u005a'
  | '\u005f'

```

```
| '\u0061'..''\u007a'  
| '\u00c0'..''\u00d6'  
| '\u00d8'..''\u00f6'  
| '\u00f8'..''\u00ff'  
| '\u0100'..''\u1fff'  
| '\u3040'..''\u318f'  
| '\u3300'..''\u337f'  
| '\u3400'..''\u3d2d'  
| '\u4e00'..''\u9fff'  
| '\uf900'..''\ufaff'  
;
```

JavaIDDigit

```
: '\u0030'..''\u0039'  
| '\u0660'..''\u0669'  
| '\u06f0'..''\u06f9'  
| '\u0966'..''\u096f'  
| '\u09e6'..''\u09ef'  
| '\u0a66'..''\u0a6f'  
| '\u0aee'..''\u0aef'  
| '\u0b66'..''\u0b6f'  
| '\u0be7'..''\u0bef'  
| '\u0c66'..''\u0c6f'  
| '\u0ce6'..''\u0cef'  
| '\u0d66'..''\u0d6f'  
| '\u0e50'..''\u0e59'  
| '\u0ed0'..''\u0ed9'  
| '\u1040'..''\u1049'  
;
```

2.7 Literals

literal

```
: null-literal  
| boolean-literal  
| string-literal  
| character-literal  
| numeric-literal  
;
```

2.7.1 Null Literals

```
null-literal
  : 'null'
  ;
```

2.7.2 Boolean Literals

```
boolean-literal
  : 'true'
  | 'false'
  ;
```

2.7.3 String Literals

```
string-literal
  : '"' ( escape-sequence | ~('\|\'|'"') )* '"'
  ;
```

2.7.4 Character Literals

```
character-literal
  : '\'' ( escape-sequence | ~('\|\'|'\|\'') ) '\''
  ;
```

2.7.5 Numeric Literals

```
numeric-literal
  : hex-literal
  | octal-literal
  | decimal-literal
  | floating-point-literal
  ;
```

```
hex-literal
  : '0' ('x'|'X') hex-digit+ integer-type-suffix?
  ;
```

```
decimal-literal
  : ('0' | '1'..'9' '0'..'9'*) integer-type-suffix?
  ;
```

```
octal-literal
```

```

    : '0' ('0'..'7')+ integer-type-suffix?
    ;

hex-digit
    : ('0'..'9'|'a'..'f'|'A'..'F')
    ;

floating-point-literal
    : ('0'..'9')+ '.' ('0'..'9')* exponent? float-type-suffix?
    | '.' ('0'..'9')+ exponent? float-type-suffix?
    | ('0'..'9')+ exponent float-type-suffix?
    | ('0'..'9')+ float-type-suffix
    ;

integer-type-suffix : ( 'l' | 'L' ) ;
float-type-suffix  : ('f'|'F'|'d'|'D') ;
exponent          : ('e'|'E') ('+ '|'-' )? ('0'..'9')+ ;

```

2.7.6 Escape Sequences

```

escape-sequence
    : '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\\'|'\\')
    | unicode-escape
    | octal-escape
    ;

octal-escape
    : '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    | '\\' ('0'..'7') ('0'..'7')
    | '\\' ('0'..'7')
    ;

unicode-escape
    : '\\' 'u' hex-digit hex-digit hex-digit hex-digit
    ;

```

2.8 Compilation unit

A DISPEL script is a compilation unit; it consist of import declarations, function declarations and block statements.

```

compilation-unit
  : import-declaration* function-declaration* statement-block*
  ;

```

2.9 Import declarations

DISPEL allows reusing existing functions by importing them before use. These import declarations are directives to the DISPEL compiler, which will facilitate the compiler in finding the reusable components. Import declarations can either import a specific function, or import all of the functions inside a package using the wild-card, '*'. The imported function (or functions, when using wild-card) must be qualified, that is they should be prefixed by the identifier of the package in which they belong. Packages are identified using a dot, '.', separated list of identifiers.

```

import-declaration
  : 'use' qualified-name ('.' '*')? ';'
  ;

```

```

qualified-name
  : identifier ('.' identifier)*
  ;

```

2.10 Statement blocks

DISPEL controls variable scope by using statement blocks.

```

statement-block
  : '{' statement* '}'
  ;

```

2.11 Statements

A statement express an operation. They are terminated using the semicolon, ';'.

```

statement
  : variable-declaration ';'
  | connection-declaration ';'
  | connect-statement ';'
  | conditional-construct
  | loop-construct
  | jump-statement ';'
  | throw-exception ';'

```

```
| null-statement ';'
;
```

```
null-statement
: ';'
;
```

2.11.1 Variable declaration

```
variable-declaration
: type variable-declarator (',' variable-declarator)*
;
```

```
variable-declarator
: identifier ('=' variable-initializer)?
;
```

2.11.2 Variable initialisation

```
variable-initializer
: array-initializer
| expression
;
```

```
array-initializer
: '{' (variable-initializer (',' variable-initializer)* (',')? )? '}'
;
```

2.11.3 Connection declaration

```
connection-declaration
: 'Connection' identifier
;
```

2.11.4 Create connection

```
connect-statement
: output-interface '=>' input-interface
| literal-sequence '=>' input-interface
;
```

2.11.5 Conditional construct

```
conditional-construct
  : 'if' boolean-expression statement-block ('else' statement-block)?
  | 'switch' expression '{' case-statement-block* default-statement-block* '}'
  ;
```

2.11.6 Loop constructs

```
loop-construct
  : 'for' '(' for-control ')' statement-block
  | 'while' boolean-expression statement-block
  | 'do' statement-block 'while' boolean-expression ';'
  ;
```

```
for-control
  : enhanced-for-control
  | for-initialise? ';' expression? ';' for-update?
  ;
```

```
enhanced-for-control
  : type identifier ':' expression
  ;
```

```
for-initialise
  : variable-declaration
  | assignment (',' assignment)*
  ;
```

```
for-update
  : assignment (',' assignment)*
  ;
```

```
repeat
  : 'repeat' expression 'of' repeat-expression
  | 'repeat' 'enough' 'of' repeat-expression
  ;
```

```
repeat-expression
  : expression
  | tuple-constant
  | list
```

```
| '(' sequenceExpressionList ')'  
;
```

2.11.7 Jump statement

```
jump-statement:  
  : 'return' expression?  
  | 'break' identifier?  
  | 'continue' identifier?  
  ;
```

2.11.8 Exception handling

```
throw-exception  
  : 'throw' expression  
  ;
```

2.11.9 Assignment statement

```
assignment  
  : conditionalExpression assignment-operator expression  
  ;
```

```
qualifiedName  
  : identifier ('.' identifier)*  
  ;
```

2.12 The type system

```
type  
  : identifier array-declaration*  
  | primitive-type array-declaration*  
  | processing-element-type array-declaration*  
  | 'void'  
  ;
```

```
processing-element-type  
  : processing-element  
  | processing-element-instance  
  ;
```

```
array-declaration
  : '[' ' ' ]'
  ;
```

2.12.1 Primitive data type

```
primitive-type
  : 'Boolean'
  | 'Integer'
  | 'Real'
  | 'String'
  | 'Identifier'
  ;
```

2.13 Expression

```
parenthesised-expression
  : '(' expression ')'
  ;
```

```
expression-list
  : expression (',' expression)*
  ;
```

```
expression
  : boolean-expression assignment-operator expression
  | boolean-expression
  ;
```

```
assignment-operator
  : '='
  | '+='
  | '-='
  | '*='
  | '/='
  | '&='
  | '|='
  | '^='
  | '%='
  ;
```

```
boolean-expression
  : boolean-or-expression
  ;
```

```
boolean-or-expression
  : boolean-and-expression ( '||' boolean-and-expression )*
  ;

boolean-and-expression
  : equality-expression ( '&&' equality-expression )*
  ;

equality-expression
  : relational-expression ( ('==' | '!=') relational-expression )*
  ;

relational-expression
  : additive-expression ( relational-operator additive-expression )*
  ;

relational-operator
  : '<='
  | '>='
  | '<'
  | '>'
  ;

additive-expression
  : multiplicative-expression ( ('+' | '-') multiplicative-expression )*
  ;

multiplicative-expression
  : unary-expression ( ( '*' | '/' | '%' ) unary-expression )*
  ;

unary-expression
  : '+' unary-expression
  | '-' unary-expression
  | '++' unary-expression
  | '--' unary-expression
  | unary-expression-not-plus-minus
  ;

unary-expression-not-plus-minus
  : '~' unary-expression
  | '!' unary-expression
```

```

| type-casting
| primary ('++'|'--')?
;

```

```

type-casting
: '(' primitive-type ')' unary-expression
| '(' (type | expression) ')' unary-expression-not-plus-minus
;

```

```

primary
: parenthesised-expression
| literal
| 'new' creator
| processing-element-instance-creator
| processing-element-creator
| identifier ('[' expression ']')*
| function-call ('[' expression ']')*
;

```

2.14 Functions

2.14.1 Calling a function

```

function-call
: identifier arguments
;

```

```

arguments
: '(' expression-list? ')'
;

```

2.14.2 Creating a function

```

function-declaration
: 'function' identifier formal-parameters type function-body
;

```

```

formal-parameters
: '(' formal-parameter-declaration? ')'
;

```

```

formal-parameter-declaration
: type variable-declarator-identifier (',' formal-parameter-declaration)?

```

```

    | 'Connection' variable-declarator-identifier (',' formal-parameter-declaration)?
    | type '...' variable-declarator-identifier
    ;

```

```

function-body
    : statement-block
    ;

```

2.15 Creator

```

creator
    : created-name arguments?
    | created-name array-creator-rest
    ;

```

```

created-name
    : identifier
    | primitive-type
    ;

```

```

array-creator-rest
    : '[' additive-expression ']' ( '[' additive-expression ']' )*
    ;

```

2.16 Sequences

```

sequence
    : '|-' sequence-expression-list '-|'
    ;

```

```

sequence-expression-list
    : sequence-expression (',' sequence-expression)*
    ;

```

```

sequence-expression
    : expression
    | list
    | tuple-constant
    | repeat
    ;

```

```

list
    : '[' sequence-expression-list ']'
    ;

```

2.17 Tuples

```
tuple-constant-list
  : tuple-constant (',' tuple-constant)*;
```

```
tuple-constant
  : '<' expression? (',' expression)* '>'
  ;
```

2.18 Input and output interfaces

```
output-interface
  : ( processing-element-instance '.' )? identifier ('[' expression ']')?
  ;
```

```
input-interface
  : ( processing-element-instance '.' )? identifier ('[' expression ']')?
  ;
```

2.19 Processing elements

2.19.1 Declaring a processing element

```
processing-element
  : 'PE' '(' input-tuple '=>' output-tuple ')'
  ;
```

```
input-tuple
  : '<' pe-input? (',' pe-input)* '>'
  ;
```

```
output-tuple
  : '<' pe-output? (',' pe-output)* '>'
  ;
```

2.19.2 Creating a processing element instance

```
processing-element-instance
  : 'new' creator
  | identifier ('[' expression ']')*
  | function-call
  ;
```

```
processing-element-instance-creator
  : 'new' processing-element
  ;

processing-element-creator
  : processing-element
  ;

processing-element-instance
  : 'PEI' '(' input-tuple '=>' output-tuple ')'
  ;

pe-input
  : 'Connection' identifier (',' identifier)* ('=' input-interface)?
  ;

pe-output
  : 'Connection' identifier (',' identifier)* ('=' output-interface)?
  ;
```

3 Conclusion and Future Work

The concrete syntax presented in this deliverable covers the essential elements of DISPEL. This is the first attempt to define the syntax systematically and as such it needs to be verified against real use cases. This is being done in ADMIRE Workpackage 6, specifically in the ACRM churn prediction scenario and in the ORAVA flood scenario, where the DISPEL syntax is used to code requests submitted from Workbench tools to the ADMIRE Gateways. Any difficulties or additional requirements will be reported back and will result in updates of the DISPEL abstract model as well as this specification.