

ART reference manual

Adrian Johnstone

`a.johnstone@rhul.ac.uk`

August 19, 2025

Department of Computer Science
Egham, Surrey TW20 0EX, England

Contents

1	Getting started	1
1.1	Downloading and first run	1
1.2	Using the ART command line interface	3
1.3	Using JavaFX with ART	3
1.4	Using the Integrated Development Environment	4
2	ART script language reference	5
2.1	The ART script interpreter	5
2.2	The ART pipeline	5
2.3	Text, characters and letters	6
2.4	Abbreviations	8
2.5	Context free grammar rules	8
2.6	Choose rules	8
2.7	Rewrite rules	8
2.8	Directives	8
2.9	Lexical builtins	8
3	The ART value system	11
3.1	ART value plugins	15
4	Script messages and tracing	17
4.1	Script messages	17

4.2	Trace messages	18
4.3	Script messages with remedial actions	18

Chapter 1

Getting started

ART is a software tool for developers of programming language interpreters and compilers which provides four core technologies: generalised parsing, ambiguity management using *choosers*, term rewriting and attribute evaluation.

ART supports a design style which we call *Ambiguity Retained Translation* (hence the name) in which multiple interpretations of a program text are allowed to co-exist rather than forcing each phase of a translator to output a single interpretation. So, for instance, decisions on whether an identifier in ANSI-C is a type name or a variable name can be delayed until a full program analysis is available.

The *ART bookshelf* is a set of documents comprising:

- ◇ **artRef** Installation instructions and a reference guide to the ART script language and the value system.
- ◇ **artTut** A tutorial guide to ART, showing how to implement simple language interpreters using either Structural Operational Semantics (SOS)-style rewriting, or attribute-action systems.
- ◇ **artLab** The laboratory guide used in the Royal Holloway undergraduate course *Software Language Engineering*
- ◇ **artInt** A guide for researchers and developers to the internals of ART, describing algorithms and their implementations.

The most recent versions of these documents may be downloaded from

<https://github.com/AJohnstone2007/ART/tree/main/doc>

1.1 Downloading and first run

1. ART is written in Java; therefore an up-to-date Java installation is required. At the time of writing, the UK Oracle download page for Java is at

<https://www.oracle.com/uk/java/technologies/downloads/>

Select and install the appropriate version for your operating system.

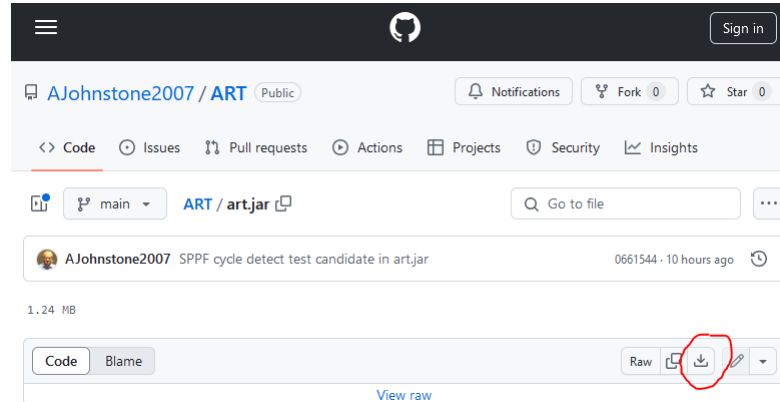
Other Java implementations are available and locatable via search engines.

2. Make a work directory, the location of which we shall call *artwork*.

Download the `art.jar` file by opening a Web browser on:

<https://github.com/AJohnstone2007/ART/blob/main/art.jar>

Click the GitHub download button (circled in red below) to download a copy of `art.jar` to your work directory *artwork*.



3. Test the download and your Java installation by opening a command window, changing your directory to *artwork* and typing the command

```
java -jar art.jar
```

The expected output is a version number, a build timestamp and summary usage information which will look like this:

```
ART 5_0_241 2024-11-01 08:12:44
Usage:
...
```

4. Instead of the ART message you may see something like this:

```
Class has been compiled by a more recent version of the Java Environment
(class file version xy.0), this version of the Java Runtime only recognizes
class file versions up to pq.0.
```

This means that your Java installation is for an old version of Java, and you will need to install a current version: see step 1.

5. The official ART repository is at

<https://github.com/AJohnstone2007/ART>

It includes the latest version of this document at

<https://github.com/AJohnstone2007/ART/blob/main/doc/slewa.pdf>

and the source code under

<https://github.com/AJohnstone2007/ART/tree/main/src>

1.2 Using the ART command line interface

ART may be run from a command line by typing `java -jar art.jar` followed by zero or more arguments.

If there are no arguments, then a help message is printed. If there are arguments, they are concatenated with separating spaces into a single input specification with the following exceptions:

1. For an argument containing a single period character and ending `.art` such as `path/name.art`, the contents of the file `path/name.art` is concatenated
2. For an argument containing a single period character and ending `.xyz` such as `path/name.xyz` where `xyz` is not lower case `art`, the string `!try 'name.xyz'` is concatenated.

The input string is then passed to the ART script language interpreter.

The effect of this is that a command of the form

```
java -jar art.jar rules.art test.str
```

will run ART using the rules in `rules.art` and test using the input string in `test.str`. Multiple `xyz.art` files will be concatenated together, and each `xyz.str` file will create a new test try. ART directives and even rules can also be inserted via the command line, for instance

```
java -jar art.jar rules.art test.str !print derivation
```

1.3 Using JavaFX with ART

ART provides support for languages that display 2D and 3D graphics using JavaFX. If your application requires graphics, then you must install JavaFX via the page at <https://gluonhq.com/products/javafx/>

Running ART with JavaFX requires a very long command line because of the need to specify class and module paths, so we recommend that you create an appropriate shell script (Un*x) or Windows batch file:

A useful Windows batch script `art.bat` contains this line:

```
java --module-path %jfxHome%\lib --add-modules javafx.controls
    -cp .;%artHome%\art.jar;%artHome%\richtextfx.jar
    uk.ac.rhul.cs.csle.art.ART %*
```

where `%jfxhome%` is the name of an environment variable bound to the location of the Java FX modules and `%artHome%` is the location of `art.jar`.

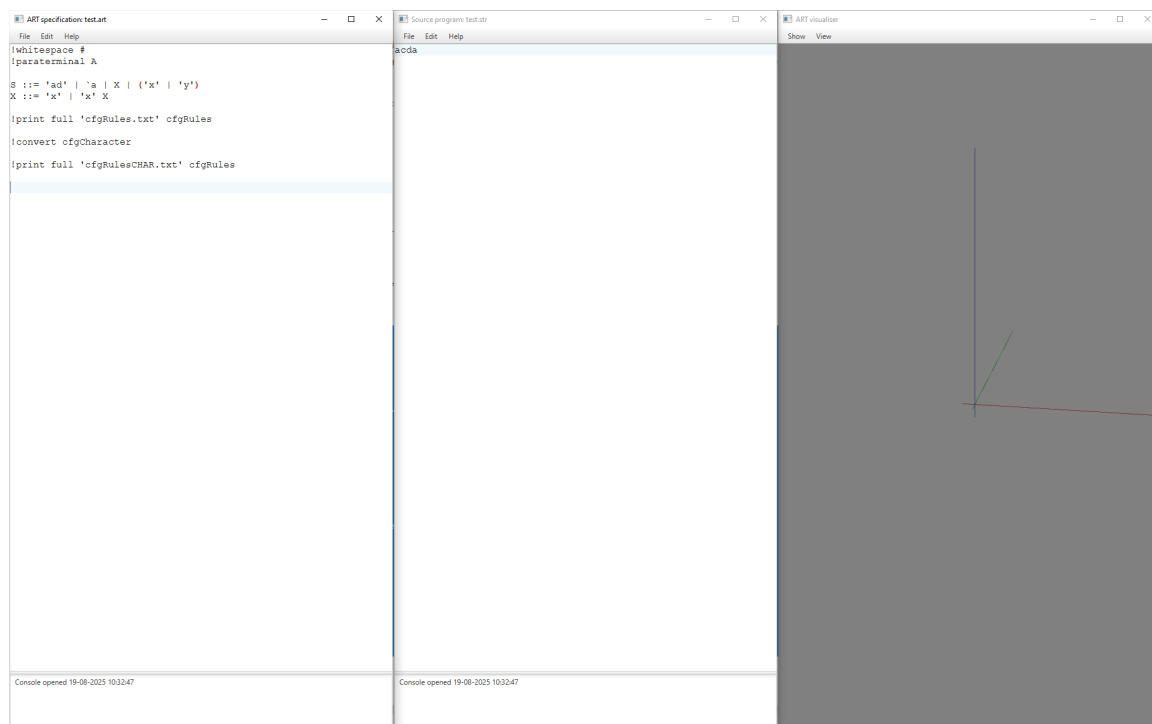
1.4 Using the Integrated Development Environment

Summer 2025: the IDE needs further development before it is ready for serious work. Please use the command line interface.

The ART jar file includes a simple Integrated Development Environment (IDE) that can aid development of language specifications. It requires JavaFX to be installed (see previous section), and in addition uses the RichTextFX editor component from <https://github.com/FXMisc/RichTextFX>. You may download a copy to your *artwork* directory from

<https://github.com/AJohnstone2007/ART/blob/main/richtextfx.jar>

In use, the IDE splits the screen into three windows: at startup the leftmost windows holds the ART script, the middle window holds the input string and the rightmost window displays visualisation information. Messages from the ART interpreter appear in the console section of the script window; messages generated by the semantics of the specified language processor appear in the console section of the middle window. These windows can be resized, moved around and iconised in the usual way.



Chapter 2

ART script language reference

An ART run interprets an *ART specification* written in the ART script language which is a sequence of four kinds of elements:

1. Context Free Grammar (CFG) rules, of the form *identifier ::= cfgExpression*
2. Choose rules, of the form *slotSet > slotSet* or *slotSet >> slotSet*
3. Term Rewrite (TR) rules, of the form *premises --- conclusion*
4. Directives, which begin with an exclamation mark **!**

The ART script language is free format, and arbitrary whitespace or comments may appear before and after each script language lexeme. The syntax specification is available from the repository at

<https://github.com/AJohnstone2007/ART/blob/main/src/uk/ac/rhul/cs/csle/art/script/scriptSpecification.art>

2.1 The ART script interpreter

During a run, ART maintains *current* versions of various *structures*, such as the current Context Free Grammar rule set, the current set of whitespace elements and the current derivation term. ART interprets a specification line by line, modifying these current structures accordingly: for instance, when a Context Free Grammar rule is encountered, it is added to the *current CFG Rule* set and similarly for Term Rewrite and Choose rules. When a **!clear cfgRules** directive is interpreted, the current CFG rule set is emptied. When a **save name cfgRules** directive is interpreted, a copy of the current CFGRules is made and bound to *name*; the directive **!recall name** make a copy of the structure bound to *name* and makes it current.

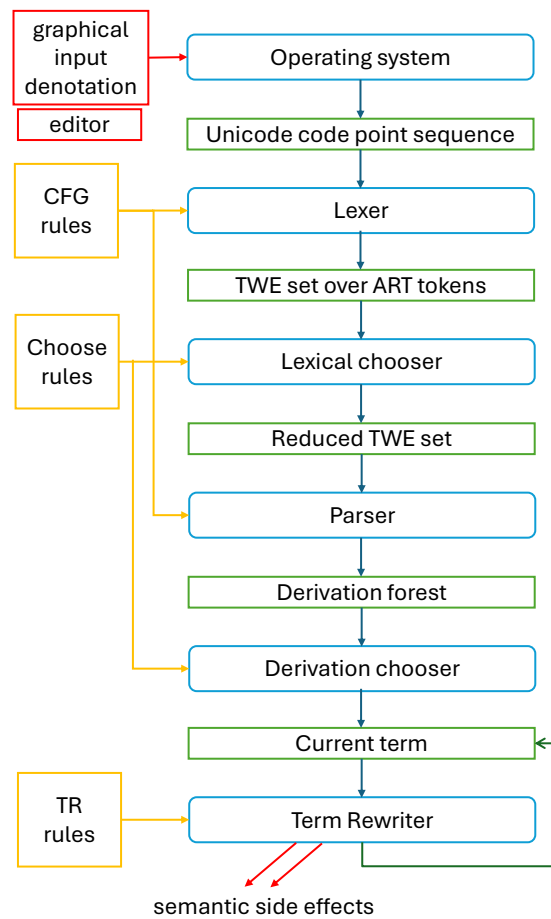
Directives are executed immediately. The most important directive is **!try** which runs the ART pipeline on a candidate input string; using the current Context Free Grammar rules and Choose rules to create a derivation term which is then rewritten and evaluated using the current Term Rewrite rules.

2.2 The ART pipeline

The pipeline comprises six processing blocks (shown as blue rounded boxes in the figure below) whose individual behaviour is parameterised by the current

rule sets in place when the `!try` is encountered, shown in yellow. The input and output data structures for each block are represented by green boxes.

The starting point is an input text, produced perhaps in a text editor. This is partitioned into lexemes using the *Lexer* which outputs a *Terminals With Extents* (TWE) set which can be then pruned using *Lexical Choose* rules; the resulting *lexicalisations* are then analysed by the *Parser* to produce a *Derivation Forest* which is pruned to produce a single *Derivation Term*. This term is then repeatedly rewritten to some final term, with semantic effects being generated as a side effects generated by ART's value system.



2.3 Text, characters and letters

ART and tools built using ART read *texts*. A text is a one dimensional sequence of *characters*, each an instance of an abstract character drawn from some character set. A character set is a finite ordered collection of characters; the position of a character in the order is called the character's *code point*, and thus a text in some particular character set may be represented as a sequence

of code points, that is a sequence of natural numbers.

A *letter* is a graphical denotation of a character that might be drawn by hand or displayed by a computer. A related collection of letters is called a *script*. Human (natural) languages are written in many scripts, for instance Greek uses letters such as $\alpha\beta\gamma$ whereas many Western languages use Latin letters such as abc. Our texts might use multiple scripts: this document is mainly written using the Latin script, but we use Greek script in some mathematical elements.

This notion of a letter is a little vague. Is the accented French e-acute \acute{e} a separate letter, or is it the letter e with some special attribute representing the accent? The conventional view is that the French alphabet has 26 letters and some accents which are not letters. On the other hand, the Swedish alphabet has 29 letters: $a \dots z$ along with \AA , \A and \O which are considered independent letters, not accented letters. Exactly what constitutes a letter is a really just a cultural convention.

A more precise way of thinking about scripts is to enumerate the *graphemes*: the set of minimal (and hence indivisible) marks that carry meaning. In the French example above, the letter e and the accent $\acute{}$ are separate graphemes but in Swedish, \AA is a single grapheme. The French \acute{e} is then thought of as a zero-width $\acute{}$ grapheme which consumes no space along a line, followed by an e grapheme which fills space on a line. These sorts of accented letters are then compounds which appear as a sequence of characters in our texts.

To successfully handle texts written in multiple scripts, we need to create a character set that has one character for every grapheme in every human language. This is exactly the goal of the Unicode Consortium: they have defined a character set comprising approximately 1.1 million code points of which at the time of writing 154,998 are used. The standard is extended annually; the 2024 revision added approximately 5,000 characters.

In addition to characters that directly represent graphemes, Unicode defines characters such as line-terminator and back-space which control very basic aspects of text display, but it does not offer encodings for things such as text colour, styles (such as italics) or font selection: these are all the province of text styling systems which build on the basic Unicode notion of a text to create *rich styled text*.

ART input texts, then, are simply sequences of Unicode characters represented as Unicode code points which an ART parser may match to other sequences of Unicode code points. How those sequences are created or displayed is of no concern to ART's algorithms.

Of course, humans prefer to use graphical denotations of a texts rather than just listing a sequence of numbers. We prefer to use an operating system's text editor to construct a graphical denotation of the text which the operating system then converts into sequence of code points that may be read into ART's input buffer. Helpfully, the operating system will also convert code points back

into graphical denotations of text when, for instance, we send code points to a printer or screen.

In this way, we can choose to think of ART as directly handling the graphical denotation of the text but that is not really true: ART only handles binary numbers; the interpretation and presentation of those numbers as lines of written text is the job of the operating system.

2.4 Abbreviations

In what follows, we make use the following abbreviations for common lexical components used in the ART script language

	Name	Pattern	Examples		
<i>ID</i>	Identifier	<code>(alpha _)(alpha _ digit)*</code>	<code>_ab</code>	<code>XYZ</code>	<code>X123</code>
<i>INT</i>	Integer	<code>(-)?digit+</code>	<code>-123</code>	<code>999</code>	<code>0</code>
<i>REAL</i>	Real	<code>(-)? (digit)+.(digit)+</code>	<code>-123.45</code>	<code>999.0</code>	<code>0.3</code>
<i>STRDQ</i>	Double-quote string	<code>"(◇)*"</code>			
<i>STRSQ</i>	Single-quote string	<code>'(◇)*'</code>			
<i>STRBR</i>	Braced string	<code>{(◇)*}</code>			
<i>STRDOL</i>	Dollar string	<code>\$(◇)*\$</code>			
<i>STRBQ</i>	Back-quote string	<code>`◇</code>			

2.5 Context free grammar rules

2.6 Choose rules

2.7 Rewrite rules

2.8 Directives

2.9 Lexical builtins

Lexical builtins are hardcoded recognisers for certain classes of substring which may be used as shorthands for common lexical patterns on the right hand side of Context Free Grammar rules. Builtin names begin with an ampersand & character.

Name	Argument	Action
<code>!prompt</code>	<i>STRDQ</i>	Print string on console and wait for carriage return
<code>!print</code>	<i>renderArg</i> list	Render the arguments as text
<code>!show</code>	<i>renderArg</i> list	Graphically visualise the arguments
<code>!trace</code>	<i>INT</i>	Set the trace level: see section ??
<code>!save</code>	<i>ID structure</i>	Bind a copy of <i>structure</i> to <i>ID</i>
<code>!recall</code>	<i>ID</i>	Make current the structure previously bound to <i>ID</i>
<code>!clear</code>	<i>structure</i>	Empties current <i>structure</i>
<code>!convert</code>	<i>ID</i> list	Apply transformations to rules
<code>!characterSet</code>	<i>STRBR</i>	Restrict input to a subset of the Unicode code points
<code>!token</code>	<i>element</i> list	
<code>!whitespace</code>	<i>element</i> list	
<code>!paraterminal</code>	<i>ID</i> list	
<code>!start</code>	<i>ID</i>	
<code>!start</code>	<i>relation</i>	
<code>!configuration</code>	<i>typedID</i> list	
<code>!terminal</code>	<i>term</i> list	
<code>!lexer</code>	<i>ID</i>	Select lexicalisation algorithm
<code>!parser</code>	<i>ID</i>	Select parsing algorithm
<code>!interpreter</code>	<i>ID</i>	Select rewriting style
<code>!mode</code>	<i>ID</i> list	Enable and disable algorithm features
<code>!try</code>	<i>STRDQ</i>	Run full pipeline on input <i>STRDQ</i>
<code>!try</code>	<i>STRDQ = term</i>	Run full pipeline on input <i>STRDQ</i> and test result
<code>!try</code>	<i>term</i>	Run rewriter only on <i>term</i> and test result
<code>!try</code>	<i>term = term</i>	Run rewriter only on first <i>term</i> and test result

Table 2.1 User directive summary

Name	Argument	Action
<code>!deleteTokens</code>		
<code>!swapTokens</code>		
<code>!derivationBreakCycles</code>		
<code>!derivationBreakCyclesRelation</code>		

Table 2.2 Experimental directive summary

Name	Examples
&CHAR.BQ	‘C
&ID	Alphanumeric Identifier
&INTEGER	123
&REAL	12.3
&STRING_BRACE	A string delimited by braces
&STRING_BRACE_NEST	A string with nested instances delimited by braces
&STRING_DOLLAR	A string delimited by dollar signs
&STRING_DQ	A string delimited by double quotes
&STRING_PLAIN_SQ	A string delimited by single quotes with no escapes
&STRING_SQ	A string delimited by single quotes
&SIMPLE_WHITESPACE	
&COMMENT_BLOCK_C	/* a C-style block comment */
&COMMENT_LINE_C	// a C-style line comment
&COMMENT_NEST_ART	(* An ART style comment (* nestable *) *)

Table 2.3 Lexical builtins

Chapter 3

The ART value system

ART provides several builtin types and operations which may be used instead of rewrite rules to perform more efficient basic arithmetic and collection operations.

3.0.1 Types

3.0.2 Value system abbreviations

Internally, all values are held as subterms whose root node is labelled with the type, and whose children contain the values.

Writing these terms out can be tiresome, so the ART front end provides a set of abbreviations that follow typical programming language conventions. This ART script exercises all of the abbreviations, showing both the ‘raw’ form used internally and the ‘cooked’ abbreviation.

```
1 !print "** The term a(b,c) with no abbreviations"
2 !print term a(b,c)
3 !prinraw term a(b,c)
4 !print "** __bool true"
5 !print term true
6 !prinraw term true
7 !print "** __bool false"
8 !print term false
9 !prinraw term false
10 !print "** __char `a"
11 !print term `a
12 !prinraw term `a
13 !print "** __intAP 1234"
14 !print term £1234
15 !prinraw term £1234
16 !print "** __int32 1234"
17 !print term 1234
18 !prinraw term 1234
19 !print "** __realAP 1234.0"
20 !print term £1234.0
```

Revised: 24 January 2025	Type	Literal	__bottom	__done	__empty	__quote	__blob	__adtprod	__adsum	__proc	__bool	__char	__intAP	__int32	__realAP	__real64	__string	__array	__list	__set	__map	Return type
Operation	Literals		__bottom	__done	__empty	__quote	__blob	__adtprod	__adsum	__proc	__bool	__char	__intAP	__int32	__realAP	__real64	__string	__array	__list	__set	__map	Return type
Equal	__eq		V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	B
Not equal	__ne		V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	V,V	B
Greater than	__gt																					B
Less than	__lt																					B
Greater than or equal	__ge																					B
Less than or equal	__le																					B
compare: return -1, 0 or +1	__comp																					B
Logical/bitwise not	__not																					B
Logical/bitwise and	__and																					B
Logical/bitwise or	__or																					B
Logical/bitwise exclusive or	__xor																					N
Shift	__shift																					V
Shift with sign extension	__sshift																					V
Rotate	__rot																					V
neg	__neg																					V
add	__add																					V
Subtract	__sub																					V
Multiply	__mul																					V
Divide	__div																					V
Remainder after division	__mod																					V
Exponentiate	__exp																					V
Cardinality	__card																					N
Insert element	__put																					V
Lookup element	__get																					V
Remove	__remove																					V
Concatenate	__cat																					A
Prefix	__prefix																					V
Suffix	__suffix																					V
Set union	__unite																					V
Set intersection	__intersect																					V
Set difference	__diff																					V
New value from deep copy	__cast																					A
Raw term operations			__bottom	__done	__empty	__quote	__blob	__adtprod	__adsum	__proc	__bool	__char	__intAP	__int32	__realAP	__real64	__string	__array	__list	__set	__map	Return type
Arity of T	__termArity		T																			__int32
T without children	__termRoot		T																			T
Call user plugin	__plugin		A, A, ...																			A

Key

A value of any type

V value column type

T any term

B bool

N __int32

Collection constructors

a a l s m

Figure 3.1 ART Value system: types, operations and signatures

```

21 !prinraw term £1234.0
22 !print "** __real64 1234.0"
23 !print term 1234.0
24 !prinraw term 1234.0
25 !print "** __array of size 3 a,b,c"
26 !print term [3 | a,b,c ]
27 !prinraw term [ 3 | a,b,c]
28 !print "** __list a,b,c"
29 !print term [ a,b,c ]
30 !prinraw term [ a,b,c]
31 !print "** empty list"
32 !print term [ ]
33 !prinraw term [ ]
34 !print "** __set a,b,c"
35 !print term { a,b,c }
36 !prinraw term { a,b,c }
37 !print "** empty set"
38 !print term { }
39 !prinraw term { }
40 !print "** __map a=p, b=q, c=r"
41 !print term { a=p, b=q, c=r }
42 !prinraw term { a=p, b=q, c=r }
43 !print "** empty map"
44 !print term {=}
45 !prinraw term {=}

```

The output from this script is:

```

1 *** Value system attached to System default plugin
2 ** The term a(b,c) with no abbreviations
3 a(b, c)
4 a(b, c)
5 ** __bool true
6 true
7 __bool(true)
8 ** __bool false
9 false
10 __bool(false)
11 ** __char `a
12 `a
13 __char(a)
14 ** __intAP 1234
15 £1234
16 __intAP(1234)
17 ** __int32 1234
18 1234
19 __int32(1234)
20 ** __realAP 1234.0
21 £1234.0

```


Constructor	Rôle	Operations
<code>__bottom</code>	Match failure	<code>__eq</code> <code>__ne</code>
<code>__done</code>		
<code>__empty</code>		
<code>__quote</code>		
<code>__blob(<i>N</i>)</code>		
<code>__proc(<i>M</i>,<i>B</i>)</code>		
<code>__adtprod(<i>V</i>,<i>N</i>)</code>		
<code>__adtsum(<i>V</i>,<i>N</i>)</code>		
<code>__bool(<i>V</i>)</code>		
<code>__char(<i>V</i>)</code>		
<code>__intAP(<i>V</i>)</code>		
<code>__int32(<i>V</i>)</code>		
<code>__realAP(<i>V</i>)</code>		
<code>__real64(<i>V</i>)</code>		
<code>__string(<i>V</i>)</code>		
<code>__array(<i>N</i>, __a(...))</code>		
<code>__list(__l(...))</code>		
<code>__set(__s(...))</code>		
<code>__map(__m(...))</code>		
<code>__map(__map(...), __m(...))</code>		

Table 3.1 ART value types and allowed operations

```

22 __realAP(1234.0)
23 ** __real64 1234.0
24 1234.0
25 __real64(1234.0)
26 ** __array of size 3 a,b,c
27 [3 | a, b, c]
28 __array(__int32(3), __a(a, __a(b, __a(c))))
29 ** __list a,b,c
30 [a, b, c]
31 __list(__l(a, __l(b, __l(c))))
32 ** empty list
33 []
34 __list
35 ** __set a,b,c
36 {a, b, c}
37 __set(__s(a, __s(b, __s(c))))
38 ** empty set
39 {}
40 __set
41 ** __map a=p, b=q, c=r
42 {a=p, b=q, c=r}
43 __map(__m(a, p, __m(b, q, __m(c, r))))
44 ** empty map
45 {=}
46 __map

```

3.0.3 Operations

3.1 ART value plugins

Constructor	Returns	Action
<code>--eq(L, R)</code>	<code>--bool</code>	value of L equal to value of R
<code>--ne(L, R)</code>	<code>--bool</code>	value of L not equal to value of R
<code>--gt(L, R)</code>	<code>--bool</code>	value of L greater than value of R
<code>--lt(L, R)</code>	<code>--bool</code>	value of L less than value of R
<code>--ge(L, R)</code>	<code>--bool</code>	value of L greater than or equal to value of R
<code>--le(L, R)</code>	<code>--bool</code>	value of L less than or equal to value of R
<code>--comp(L, R)</code>	<code>--int32</code>	if $L < R$ then -1 else if $L > R$ then +1 else 0
<code>--not(L)</code>	$T(L)$	Logical or bitwise inversion
<code>--and(L, R)</code>	$T(L)$	Logical or bitwise conjunction
<code>--or(L, R)</code>	$T(L)$	Logical or bitwise disjunction
<code>--xor(L, R)</code>	$T(L)$	Logical or bitwise exclusive OR
<code>--shift(L, R)</code>	$T(L)$	Left shift L by R bits
<code>--sshift(L, R)</code>	$T(L)$	Right shift L by R bits, propagating zeroes
<code>--rot(L, R)</code>	$T(L)$	Right shift L by R bits, propagating sign bit
<code>--neg(L)</code>		
<code>--add(L, R)</code>		
<code>--sub(L, R)</code>		
<code>--mul(L, R)</code>		
<code>--div(L, R)</code>		
<code>--mod(L, R)</code>		
<code>--exp(L, R)</code>		
<code>--card(L)</code>		
<code>--put(L, K, V)</code>		
<code>--get(L, R)</code>		
<code>--remove(L, K)</code>		
<code>--cat(L, R)</code>		
<code>--prefix(L, R)</code>		
<code>--suffix(L, K)</code>		
<code>--unite(L, R)</code>		
<code>--intersect(L, R)</code>		
<code>--diff(L, R)</code>		
<code>--cast(L, R)</code>		

Table 3.2 ART value operations

Chapter 4

Script messages and tracing

Messages from ART come in four categories: *script messages* which report progress and problems with the execution of scripts; *trace messages* which report the detailed progress of parsers and rewriters; *script outputs* in response to the directives `!print` and `!show`; and *user outputs* generated by the language semantics specified in the script.

In this chapter we discuss the first two categories. Outputs from directives are documented in section 2.8 and user outputs are, by their nature, application specific.

4.1 Script messages

ART emits script messages at one of four *severity levels*: fatal, error, warning, and informational. A *fatal* message indicates that the internal integrity of ART is compromised (such as by running out of memory) and as a result ART is shutting down and terminating.

An *error* message indicates that ART cannot proceed with the current directive, and has terminated execution of the associated action. Output files may be left in an inconsistent state. The script should be modified to correct the error.

A *warning* message indicates that ART has detected an anomaly of some sort, but is continuing to execute anyway; we recommend that scripts are modified to run without warnings.

An *informational* message is a simple progress report requiring no user action.

Output of messages is controlled by an internal variable that may be set with the `!errorLevel` directive which takes an integer in the range 0–3. **The default error level is 3.**

Error level	Messages displayed
0	Only fatal messages
1	Error and fatal
2	Warning, error and fatal
3	Informational, warning, error and fatal messages

4.2 Trace messages

ART can give detailed progress reports during parsing and rewriting under the control of an internal variable that may be set with the `!traceLevel` directive which takes an integer in the range of 0–9. As above, levels are cumulative in the sense that for level n , all messages for levels 0– n will be displayed.

Trace level	Parser	Rewriter
0	(Silent)	(Silent)
1	Parser accept	
2		Rewriter termination
3		Step number
4		Rewrite attempt
5		Rule selection
6		Premises
7		Bindings
8	Lexical match	
9	Stack activity	
10	Derivation activity	
11	Task activity	

The default trace level is 3.

4.3 Script messages with remedial actions