ART reference manual

Adrian Johnstone
a.johnstone@rhul.ac.uk

August 23, 2025

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

Contents

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.

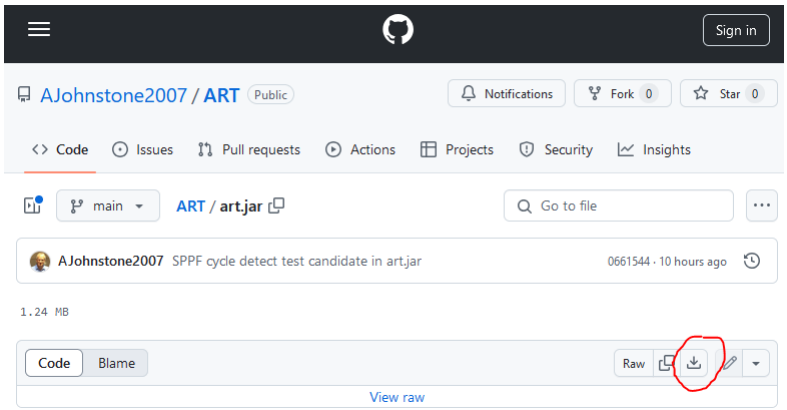
   Other Java implementations are available and locatable via search engines.

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 the ART bookshelf documents at

   https://github.com/AJohnstone2007/ART/blob/main/doc

   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 the first argument is `fx`, run as a Java FX application (see section 1.3).

If the first argument is `ide`, open the ART Integrated Development Environment (see section 1.4).

The rest of the arguments 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 and `%artHome%` is the location of `art.jar`.

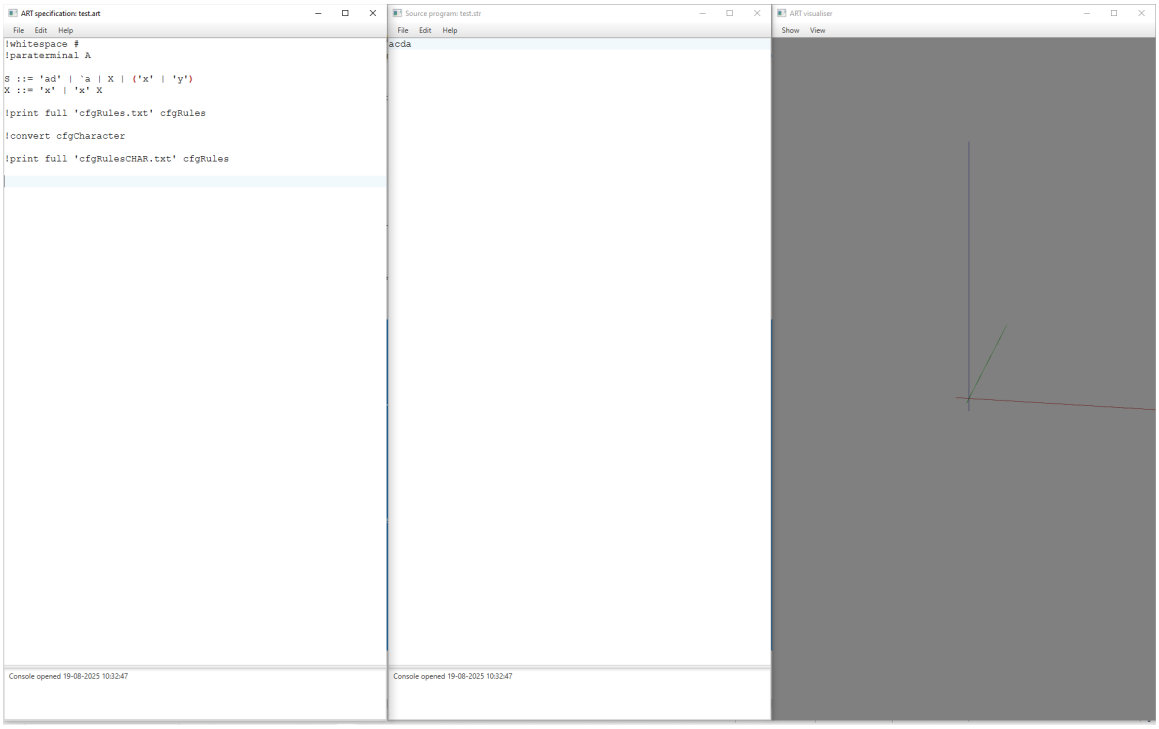
1.4 Using the Integrated Development Environment

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

The script language

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 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.

2.2 !try and the ART pipeline

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.

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

Name	Rôle
CFG rules	
Choose rules	
TR rules	
whitespaces	
paraterminals	
characters	
start nonterminal	
derivations	
term	
start relation	
final terms	

Table 2.1 ART structures

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 analysd 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

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 *é* 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*...*z* along with *å, ä and ö* which are considered independent

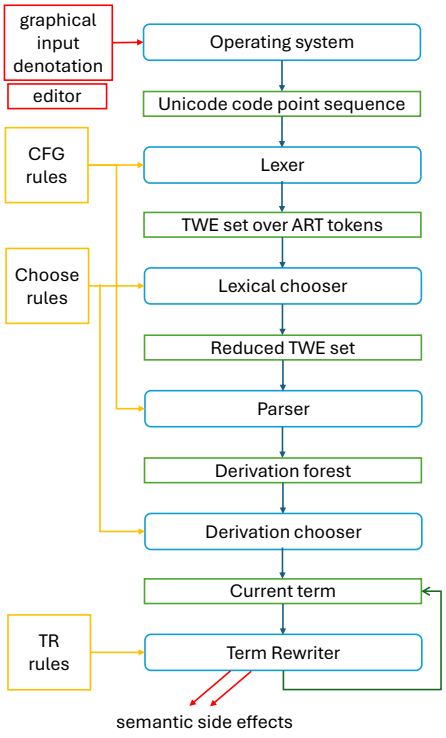


Figure 2.1 The ART pipeline

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 ´ are separate graphemes but in Swedish, *å* is a single grapheme. The French *é* is then thought of as a zero-width ´ 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

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

In strings, the symbol ◇ denotes any printable letter *except* for the closing delimiter for that style of string along with the escape sequences listed in Table 2.3.

Table 2.2 Lexical elements of the ART script language

rich or *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, stored or displayed is of no concern to ART's algorithms.

Of course, humans prefer to use graphical denotations of a text 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 Script language lexical elements

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

2.5 Directives

A directive instructs the ART script interpreter to take some immediate action. A summary of the available directives is shown in Table 2.4 which contains links to the detailed descriptions below.

There is also a set of experimental directives which test out new features or support aspects of our research papers. These experimental directives are not

Sequence	code point name
`\b`	back space
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	horizontal tab
`\"`	double quote
`\'`	single quote
`\}`	close brace
`\$`	dollar
`\uWXYZ`	Unicode BMP (16-bit) code point where WXYZ is a four-digit hex number
`\vWXYZ`	full Unicode code point where UVWXYZ is a six-digit hex number

Table 2.3 Escape sequences

intended for general use, and may be removed or may change their behaviour in future versions; they are listed Table 2.5 for completeness.

2.5.1 Displays: !prompt !print !show !trace !error

!prompt

!print

!show

!trace

!error

2.5.2 Structures: !save !recall !clear !convert

!save

!recall

!clear

!convert

Name	Argument	Action
!prompt	*STRDQ*	Print string on console and wait for carriage return
!print	*structure* list	Render structures as text
!show	*structure* list	Graphically visualise structures
!trace	*INT*	Set the trace threshold: see section 4.2
!error	*INT*	Set the error threshold: see section 4.1
!save	*ID structure*	Bind a copy of *structure* to *ID*
!recall	*ID*	Make current a copy of the structure previously bound to *ID*
!clear	*structure*	Empties current *structure*
!convert	*ID* list	Apply transformations to structures
!character	*STRBR*	Restrict input to a subset of the Unicode code points
!token	*element* list	Enumerate lexical tokens
!whitespace	*element* list	Clear current whitespace set and add elements from list
!paraterminal	*ID* list	Clear current paraterminal set and add nonterminals from list
!configuration	*typedID* list	Clear current configuration tuple and add typed terms
!final	*term* list	Clear current TR final term list and add terms from the list
!lexer	*ID*	Select lexicalisation algorithm
!parser	*ID*	Select parsing algorithm
!interpreter	*ID*	Select interpreter
!mode	*ID* list	Enable and disable algorithm features
!start	*ID*	Set the start nonterminal for the CFG rules
!start	*relation*	Set the start relation for the TR rules
!try	*STRDQ*	Run full pipeline on input *STRDQ*
!try	*STRDQ* = *term*	Run full pipeline on input *STRDQ* and test result
!try	*term*	Run rewriter only on *term*
!try	$term_1$ = $term_2$	Run rewriter only on $term_1$ and test result against $term_2$

Table 2.4 Directive summary

Name	Argument	Action
!deleteTokens	*INT*	Remove *INT* tokens from centre of lexicalisation
!swapTokens	*INT*	Reverse order of *INT* tokens from centre of lexicalisation
!breakCycles	*none*	Break cycles in derivations using SLE25 paper algorithm
!breakCyclesRelation	*none*	Generate cycle break relation using SLE25 paper operations

Table 2.5 Experimental directive summary

2.5.3 CFG: !characters !token !whitespace !paraterminal

!character

!token

!whitespace

!paraterminal

2.5.4 TR: !configuration !final

!configuration

!final

2.5.5 Algorithms: !lexer !parser !interpeter !mode

!lexer

!parser

!interpreter

!mode

2.5.6 Pipeline activation: !start !try

!start

!try

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 2.6 Lexical builtins

2.6 Context free grammar rules

2.6.1 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.

2.6.2 GIFT annotations

2.6.3 Attributes and actions

2.7 Choose rules

2.8 Rewrite rules

Chapter 3

The 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.

3.1 Types

3.2 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  !printraw term a(b,c)
4  !print "** __bool true"
5  !print term true
6  !printraw term true
7  !print "** __bool false"
8  !print term false
9  !printraw term false
10 !print "** __char `a"
11 !print term `a
12 !printraw term `a
13 !print "** __intAP 1234"
14 !print term £1234
15 !printraw term £1234
16 !print "** __int32 1234"
17 !print term 1234
18 !printraw term 1234
19 !print "** __realAP 1234.0"
20 !print term £1234.0
21 !printraw term £1234.0
```

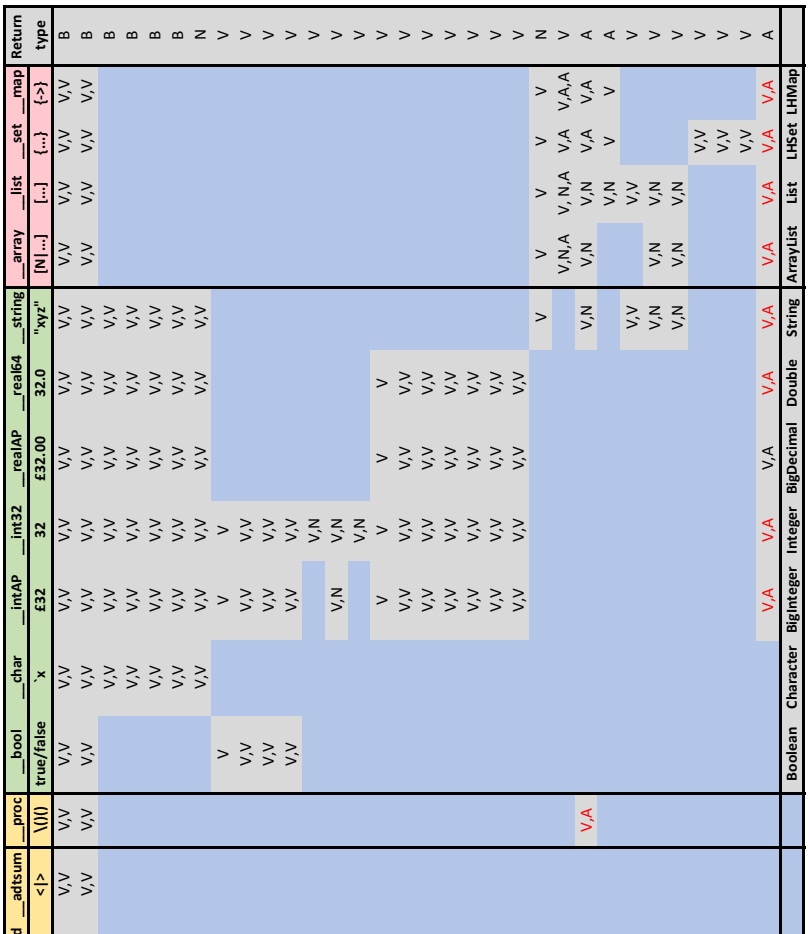


Figure 3.1 ART Value system: types, operations and signatures

```
!print "** __real64 1234.0"
!print term 1234.0
!printraw term 1234.0
!print "** __array of size 3 a,b,c"
!print term [3 | a,b,c ]
!printraw term [ 3 | a,b,c]
!print "** __list a,b,c"
!print term [ a,b,c ]
!printraw term [ a,b,c]
!print "** empty list"
!print term [ ]
!printraw term [ ]
!print "** __set a,b,c"
!print term { a,b,c }
!printraw term { a,b,c }
!print "** empty set"
!print term { }
!printraw term { }
!print "** __map a=p, b=q, c=r"
!print term { a=p, b=q, c=r }
!printraw term { a=p, b=q, c=r }
!print "** empty map"
!print term {=}
!printraw term {=}
```

The output from this script is:

```
*** Value system attached to System default plugin
** The term a(b,c) with no abbreviations
a(b, c)
a(b, c)
** __bool true
true
__bool(true)
** __bool false
false
__bool(false)
** __char `a
`a
__char(a)
** __intAP 1234
£1234
__intAP(1234)
** __int32 1234
1234
__int32(1234)
** __realAP 1234.0
£1234.0
__realAP(1234.0)
```

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

Table 3.1 ART value types and allowed operations

```
** __real64 1234.0
1234.0
__real64(1234.0)
** __array of size 3 a,b,c
[3 | a, b, c]
__array(__int32(3), __a(a, __a(b, __a(c))))
** __list a,b,c
[a, b, c]
__list(__l(a, __l(b, __l(c))))
** empty list
[]
__list
** __set a,b,c
{a, b, c}
__set(__s(a, __s(b, __s(c))))
** empty set
{}
__set
** __map a=p, b=q, c=r
{a=p, b=q, c=r}
__map(__m(a, p, __m(b, q, __m(c, r))))
** empty map
{=}
__map
```

3.3 Operations

3.4 ART plugins

Constructor	Returns	Action
__eq(L, R**)**	**__bool**	value of L equal to value of R
__ne(L, R**)**	**__bool**	value of L not equal to value of R
__gt(L, R**)**	**__bool**	value of L greater then value of R
__lt(L, R**)**	**__bool**	value of L less than value of R
__ge(L, R**)**	**__bool**	value of L greater that or equal to value of R
__le(L, R**)**	**__bool**	value of L less than or equal to value of R
__comp(L, R**)**	**__int32**	if $L < R$ then -1 else if $L > R$ then +1 else 0
__not(L**)**	$T(L)$	Logical or bitwise inversion
__and(L, R**)**	$T(L)$	Logical or bitwise conjunction
__or(L, R**)**	$T(L)$	Logical or bitwise disjunction
__xor(L, R**)**	$T(L)$	Logical or bitwise exclusive OR
__shift(L, R**)**	$T(L)$	Left shift L by R bits
__sshift(L, R**)**	$T(L)$	Right shift L by R bits, propagating zeroes
__rot(L, R**)**	$T(L)$	Right shift L by R bits, propagating sign bit
__neg(L**)**		
__add(L, R**)**		
__sub(L, R**)**		
__mul(L, R**)**		
__div(L, R**)**		
__mod(L, R**)**		
__exp(L, R**)**		
__card(L**)**		
__put(L, K, V**)**		
__get(L, R**)**		
__remove(L, K**)**		
__cat(L, R**)**		
__prefix(L, R**)**		
__suffix(L, K**)**		
__unite(L, R**)**		
__intersect(L, R**)**		
__diff(L, R**)**		
__cast(L, R**)**		

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 2.5 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 Error messages with remedial actions