ART reference manual

Adrian Johnstone
a.johnstone@rhul.ac.uk

September 15, 2025

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

Contents

1 Getting started **1**

1.1 Documentation 1

1.2 Downloading and first run 1

1.3 Using the ART command line interface 3

1.4 Using JavaFX with ART 3

1.5 Using the Integrated Development Environment 4

2 Script language fundamentals **5**

2.1 Script language lexical elements 5

2.2 Script structures 6

2.3 The `!try` pipeline 7

2.4 Input text, characters and letters 8

2.5 In-band and out-of-band code points 9

3 Directives **11**

3.1 `!characterSet` *STRBR* 12

3.2 `!clear` *ID* LIST 12

3.3 `!configuration` ID_E`:`ID_T LIST 13

3.4 `!convert` *ID* LIST 13

3.5 `!errorLevel` *INT* 13

3.6 `!final` *INT* 13

3.7 `!generate` *INT ID* 13

3.8 !generate *ID* 13

3.9 !interpreter *interpreterAlg* 13

3.10 !lexer *lexerAlg* 13

3.11 !mode *ID* LIST 13

3.12 !paraterminal *cfgElement* LIST 13

3.13 !parser *parserAlg* 13

3.14 !print *ID* LIST 13

3.15 !prompt *STRDQ* 15

3.16 !recall *ID* 15

3.17 !save ID_1 ID_2 15

3.18 !show *ID* LIST 15

3.19 !start *ID* 15

3.20 !start *relation* 15

3.21 !support *action* LIST 15

3.22 !token *cfgElement* LIST 15

3.23 !traceLevel *INT* 15

3.24 !try *STRDQ* 15

3.25 !try *STRDQ* = *term* 15

3.26 !try *term* 15

3.27 !try *term* $term_1$ = $term_2$ 15

3.28 !whitespace *cfgElement* LIST 15

4 Context free grammar rules 17

5 Choose rules 19

6 Rewrite rules 21

7 The value system 23

7.1 Types 23

7.2 Value system abbreviations 23

7.3 Operations 27

7.4 ART plugins 27

8 Script messages and tracing 29

8.1 Script messages 29

8.2 Trace messages 30

8.3 Error messages with remedial actions 30

List of Tables

2.1 Lexical elements of the ART script language 5

2.2 Escape sequences 6

2.3 Script structures 6

3.1 Directive summary 11

3.2 Experimental directive summary 12

3.3 `!print` and `!show` mode control arguments 13

3.4 `!print` and `!show` display arguments 14

4.1 Lexical builtins 18

7.1 ART value types and allowed operations 26

7.2 ART value operations 28

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.

1.1 Documentation

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 (this document).
- **artSLE** A tutorial guide to software language engineering with ART, showing how to implement 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

1.2 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.

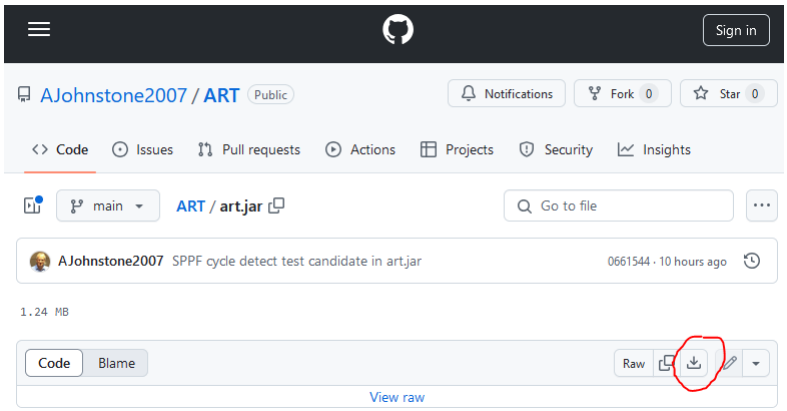
Other Java implementations are available and locatable via search engines.

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

1.3 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 1.4).

If the first argument is `ide`, open the ART Integrated Development Environment (see section 1.5).

All other 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
```

1.4 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 Windows batch file or (Un*x) shell script.

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`.

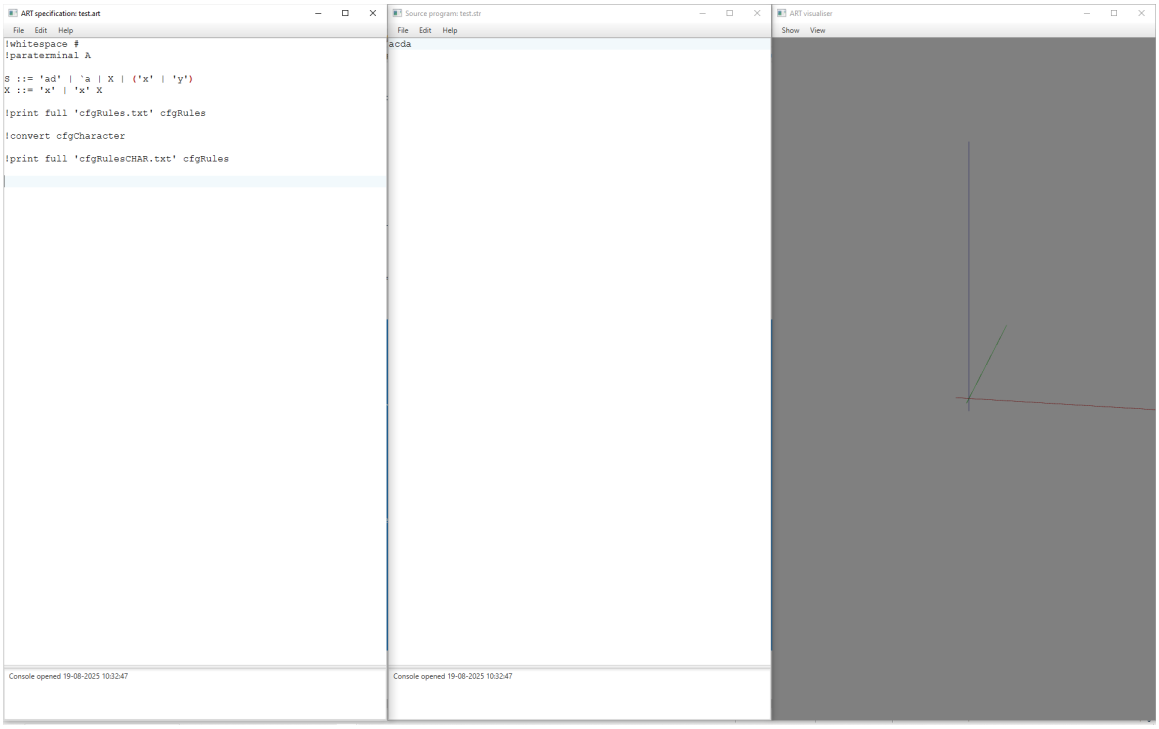
1.5 Using the Integrated Development Environment

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

Script language fundamentals

ART interprets specifications written in the ART script language. The latest ART syntax specification is available from the repository **here**.

A specification is a sequence of four kinds of phrase:

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

2.1 Script language lexical elements

The ART script language is free format, and a sequence of whitespace characters and comments may appear before and after each script language lexical element: whitespace characters are: newline, return, tab and space; comments are delimited by `(*` and `*)`, or by `//` and line end.

Language lexical elements comprise the fixed keywords and punctuation defined in the ART syntax specification, along with the elements listed in Table 2.1: comments, identifiers, real and integer literals and several styles of string .

	Name	Pattern	Examples
COMBLOCK	Block comment	`(*`◇*`*)`	`(* comment *)`
COMLINE	Line comment	`//`◇* newline	`// comment`
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 and comments, the symbol ◇ denotes any printable letter *except* for the closing delimiter, along with the escape sequences listed in Table 2.2.

Table 2.1 Lexical elements of the ART script language

Within strings, non-printing characters such as newline are discouraged. Instead, use an *escape sequence* introduced by a backslash \ as listed in Table 2.2. Any other sequence `\x` yields the character *x*.

Sequence	code point name
`\b`	back space
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	horizontal tab
`\uWXYZ`	Unicode BMP (16-bit) code point where WXYZ is a four-digit hex number
`\vUVWXYZ`	full Unicode code point where UVWXYZ is a six-digit hex number

Table 2.2 Escape sequences

2.2 Script structures

As it processes rules and directives, ART updates various structures such as the current Context Free Grammar rule set, the current set of whitespace elements and the current derivation term. The full set of structures and their function is shown in Table 2.3.

Name	Part of	Rôle	Default value
`cfgRules`	-	Parser and lexer rules	Empty
`start`	`cfgRules`	Parser start symbol	LHS of first CFG rule
`characterSet`	`cfgRules`	The set of 'in band' characters	All terminal characters
`whitespace`	`cfgRules`	The set of whitespace elements	`&SIMPLE_WHITESPACE`
`paraterminal`	`cfgRules`	The set of paraterminals	Empty
`token`	`cfgRules`	Token list	All specified terminals
`lexicalisations`	-	Lexicalisations from the most recent `!try`	Empty
`stacks`	-	Stack entries from the most recent `!try`	Empty
`derivations`	-	Derivation steps from the most recent `!try`	Empty
`tasks`	-	Task descriptors from the most recent `!try`	Empty
`chooseRules`	-	Lexical and derivation Choose rules	Longest match, literal priority
`trRules`	-	Term rewriter rules	Empty
`start`	`trRules`	Initial term rewriter relation symbol	Relation from first TR rule
`finalTerms`	`trRules`	Normal forms (terminals) of the rewrite rules	All ART values
`term`	-	Resulting term from the most recent `!try`	Null term

Table 2.3 Script structures

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 Choose rules and Term Rewrite rules.

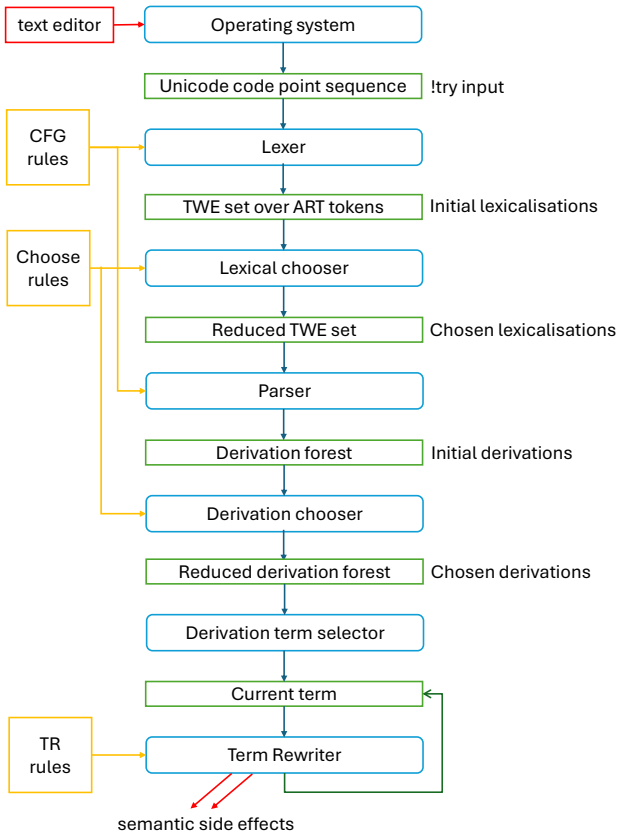
Some of these structures are really sub-structures of others. The second column

in Table 2.3 shows the parent structure for each element: a `-` in that column means that the corresponding element is a top level structure. Rule sets may be cleared, saved and restored. So, for instance, when the whitespace set is updated it is the whitespace of the current `cfgRules` that is modified; and that value will be saved and restored as part of the Conetxt Free Grammar rule set. Similarly the `finalTerms` (the irreducible terms) of a relation 'belong' to the current `trRules`, and will be saved and restored when the `trRules` are saved and restored.

2.3 The `!try` pipeline

An ART specification sets up rules and definitions for a translation run which is then triggered by a `!try` directive, which activates a pipeline made up of the seven processing blocks shown as blue rounded boxes below.

The input and output data structures for each block are represented by green boxes; each block's behaviour is parametrised by the current rule sets in place when the run starts, shown in yellow.



The starting point is an input text, produced perhaps in a text editor. The operating system delivers this to ART as a sequence of Unicode code points.

This sequence is partitioned into lexemes using the *Lexer* which outputs a *Terminals With Extents* (TWE) set containing all possible partitions of the input; these are the initial lexicalisations of the input.

This TWE set can be pruned to reduce lexical ambiguity using *Lexical Choose* rules. The resulting reduced TWE set, representing a reduced set of lexicalisations, is then analysed by the *Parser* to produce an initial set of derivation steps, arranged as a forest of derivation trees.

This forest can be pruned by the *Derivation Choose* rules to produce a reduced forest of derivation trees which is converted to a single tree (or *Current term*) by the *Derivation Term Selector*. If the reduced derivation forest contains more than one tree then an ambiguity tree node is used to gather together shared subtrees; this ensures that the output is a single tree, but at the cost of a potentially exponential increase in the space requirements.

The final stage of the pipeline is the *Term Rewriter* which repeatedly rewrites the current term to some final term, with semantic effects being generated as side effects generated by ART's value system.

The term rewriter may be configured to process Attribute-Action systems, that is a set of Context Free Grammar rules with embedded actions and value expressions that evaluate attributes associated with derivation tree nodes; there is also a specialised high-speed interpreter for these kinds of specifications.

2.4 Input text, characters and letters

Tools built using ART process *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 letters, not accented letters. Exactly what constitutes a letter is 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 *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.

2.5 In-band and out-of-band code points

The size of the complete Unicode character set presents an implementation challenge. Context Free Grammars are formally defined over finite alphabets. Now, although the Unicode code point character space is finite, it has been extended in the past, and conceivably might be extended again. More significantly, whilst the number of *defined* code points with an allocated character is fixed at any given time, it is extended annually, and already stands at over

150,000 characters. Any attempt to write an ART CFG rule that listed all possible characters would be (a) very large and (b) quickly out of date.

Does this matter? Typically programming language standards enumerate the valid character set that is used by the language proper. However, that leaves open the question of language elements that are merely carriers for character information, such as string literals and comments.

We should like programmers to be able to process character strings in any script, and add comments in their own language and script, independent of the script used to denote the programming language's keywords. Java, for instance, allows this, so if we wanted to write an ART specification for Java, then we need a way to denote the full capabilities of the Unicode standard, at a time in the future when it might have expanded.

In practice, we can simply enumerate all of the code points that are used in ART literal terminals, add in those arising from any builtins and treat that as the complete valid set for the programming language itself. We call this set of characters the *in-band* code points or the in-band character set;. we can reference it concisely in ART as `~{}`, that is as the anti-set with no members.

The set of Unicode code points that are not in the in-band set is called the *out-of-band* set; in ART specifications we denote the out-of-bound character set as `~!{}`

This is a subtle distinction, and is a possible source of confusion. Why not simply let `~{`***x***`}` match any valid Unicode character except ***x***? Well, the underlying production would have to enumerate the whole Unicode character set except ***x*** and would soon be out of date.

Instead, we only allow character sets and anti-sets over the in-band characters (which are well-defined by other components of the ART specification), and have separate wild card match for the rest of the Unicode set. Importantly, there is no way to match a subset of the out-of-band characters.

Chapter 3

Directives

A directive instructs the ART script interpreter to take some immediate action. A summary of the available directives is shown in Table 3.1 which contains links to more detailed descriptions below.

Directive	Arguments	Action
`!prompt`	*STRDQ*	Print string on console and wait for carriage return
`!traceLevel`	*INT*	Set the trace threshold: see section 8.2
`!errorLevel`	*INT*	Set the error threshold: see section 8.1
`!print`	*ID* LIST	Render structures as text
`!show`	*ID* LIST	Graphically visualise structures
`!clear`	*ID* LIST	Empties current structure *ID*
`!save`	ID_1 ID_2	Bind a copy of structure ID_2 to ID_1
`!recall`	*ID*	Make current a copy of the structure previously bound to *ID*
`!convert`	*ID* LIST	Apply transformations to structures
`!characterSet`	*STRBR*	Restrict in-band characters to a subset of the Unicode code points
`!token`	*cfgElement* LIST	Enumerate lexical tokens
`!whitespace`	*cfgElement* LIST	Replace current whitespace with elements from LIST
`!paraterminal`	*cfgElement* LIST	Replace current paraterminal set with elements from LIST
`!configuration`	ID_E : ID_T LIST	Replace current configuration with entities ID_E of type ID_T
`!final`	*term* LIST	Clear current TR final term set and add terms from LIST
`!lexer`	*lexerAlg*	Select lexicalisation algorithm - default `GLLRecogniser`
`!parser`	*parserAlg*	Select parsing algorithm - default `mGLL`
`!interpreter`	*interpreterAlg*	Select interpreter algorithm - default `eSOS`
`!generate`	*INT ID*	Generate sentential forms from CFG rules
`!generate`	*ID*	Generate compilable translator artefacts
`!support`	*action* LIST	Specify imports and globals for generated artefacts
`!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 against *term*
`!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 3.1 Directive summary

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 intended for general use, and may be removed or may change their behaviour in future versions; they are listed Table 3.2 for completeness but are not documented here.

Directive	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 3.2 Experimental directive summary

3.1 `!characterSet` *STRBR*

Restrict in-band characters to a subset of the Unicode code points.

This directive does not usually appear in user-level specifications. The intended application is to synchronise the character sets of linked grammars, in particular the lexical and parser sub-grammars generated internally by ART for use with some styles of lexer.

The effect is to define the in-band subset of Unicode to be used with the current specification. If no `!characterSet` directive is present, then the in-band character set is the set of characters that appear in terminals.

If a `!characterSet` directive is present, it is an error for there to be characters in ART terminals that are not included in the argument.

Example: `!characterSet {abcdefghijklmnopqrsturvwxyz£}`

Effect: restrict the current character set to the lower case Latin code points and the pound sterling code point.

3.2 `!clear` *ID* LIST

Empties current structure *ID*

Referring to Table 2.3, ART maintains a set of top level and sub-structures that are updated as the specification script is interpreted. A `!clear` *ID* directive (where *ID* is one of the names in column one of the table) removes updates to that structure, and returns it to the default value given in the fourth column of the table. If the structure is a top-level structure, then all of the sub-structures that are part of that structure are also returned to their default state.

Example: `!clear whitespace, trRules`

Effect: empty the whitespace component of the current Context Free Grammar rules, and remove all Term Rewrite rules along with their final terms and start relation.

Mode element	Effect on subsequent outputs
`'abc.xyz'`	Redirect output to file `abc.xyz`
`raw`	Show full term form without abbreviations
`indented`	Output each new subterm on a new, indented, line
`depth` *INT*	Replace subterms deeper than *INT* with ellipsis `...`
`indexed`	Add numerical indices
`full`	Give more detailed output: eg for `cfgRules` show first and follow sets
`plain`	Switch to plain text output
`html`	Switch to HTML styling employing elements from **artStyle.css**
`latex`	Switch to LaTeX styling employing macros from **artStyle.tex**
`markDown`	Switch to Markdown styling

Table 3.3 `!print` and `!show` mode control arguments

3.3 `!configuration` ID_E`:`ID_T LIST

3.4 `!convert` *ID* LIST

3.5 `!errorLevel` *INT*

3.6 `!final` *INT*

3.7 `!generate` *INT ID*

3.8 `!generate` *ID*

3.9 `!interpreter` *interpreterAlg*

3.10 `!lexer` *lexerAlg*

3.11 `!mode` *ID* LIST

3.12 `!paraterminal` *cfgElement* LIST

3.13 `!parser` *parserAlg*

3.14 `!print` *ID* LIST

Render as text the elements *ID* LIST. The argument list is processed left-to-right, using the operations defined in Tables 3.3 and 3.4.

Display element	Output
`version`	current ART version number and build date
`"String"`	literal within double quotes
`term` *`termLiteral`*	Output literal *term*
`cfgRules`	Current Context Free Grammar rule set
`cfgRulesLexer`	Current lexical-level Context Free Grammar rule set
`cfgRulesParser`	Current parser-level Context Free Grammar rule set
`chooseRules`	Current Choose rule set
`trRules`	Current Term Rewriter rule set
`lexicalisations`	Lexicalisations (TWE set) from most recent `!try`
`tasks`	Completed task descriptors from most recent `!try`
`stacks`	Stack nodes (GSS) from most recent `!try`
`derivations`	Derivation steps (SPPF or BSR set) from most recent `!try`
`tryTerm`	Final derivation term from most recent `!try`
`statistics`	Statistics log from most recent `!try`
`cardinalities`	Summary set sizes from most recent `!try`
`paraterminals`	Paraterminals that appear in derivations from most recent `!try`
`parasentences`	Parasentences embedded in derivations from most recent `!try`
`scriptCFGRules`	Current CFG rules for ART's script language
`scriptLexicalisations`	Lexicalisation of the current specificationt
`scriptDerivations`	Derivations of the current specification
`scriptTerm`	Final derivation term of the current specification

Table 3.4 `!print` and `!show` display arguments

3.15 `!prompt` *STRDQ*

3.16 `!recall` *ID*

3.17 `!save` ID_1 ID_2

3.18 `!show` *ID* LIST

Graphically render the elements *ID* LIST. The argument list is processed left-to-right, using the operations defined in Tables 3.3 and 3.4.

When running under the IDE, the visualisation will appear as an interactive graphics window. When not running under the IDE, or after a `file` argument, the visualisation will be written to `file` in the GraphViz `.dot` format for processing and visualisation using GraphViz.

3.19 `!start` *ID*

3.20 `!start` *relation*

3.21 `!support` *action* LIST

3.22 `!token` *cfgElement* LIST

3.23 `!traceLevel` *INT*

3.24 `!try` *STRDQ*

3.25 `!try` *STRDQ* `=` *term*

3.26 `!try` *term*

3.27 `!try` *term* $term_1$ `=` $term_2$

3.28 `!whitespace` *cfgElement* LIST

Chapter 4

Context free grammar rules

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.

GIFT annotations

Attributes and actions

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 4.1 Lexical builtins

Chapter 5

Choose rules

Chapter 6

Rewrite rules

Chapter 7

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.

7.1 Types

7.2 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
```

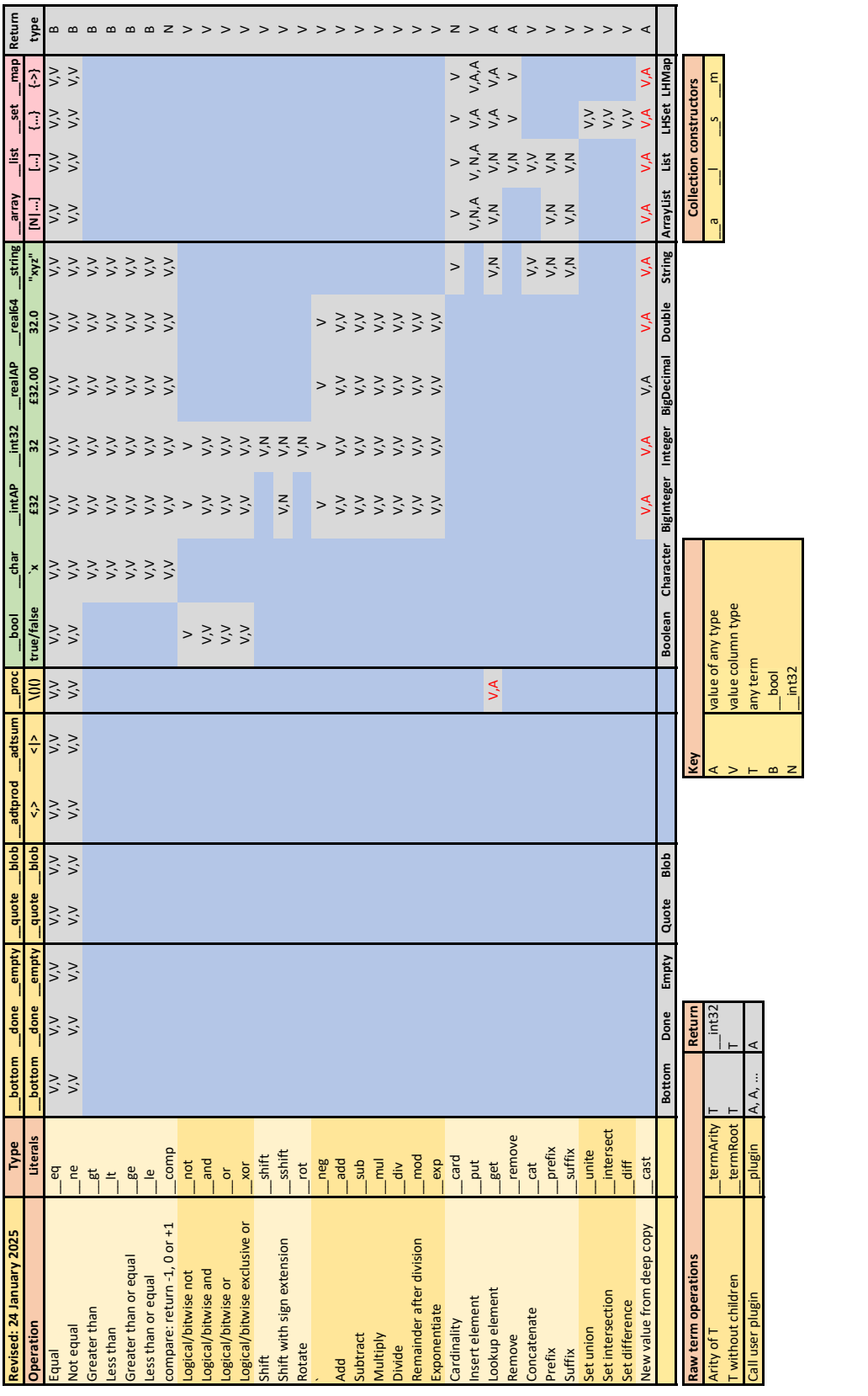


Figure 7.1 ART Value system: types, operations and signatures

```
!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 7.1 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
```

7.3 Operations

7.4 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 7.2 ART value operations

Chapter 8

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 3 and user outputs are, by their nature, application specific.

8.1 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

8.2 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	Lexer activity	
9	Stack activity	
10	Derivation activity	
11	Task activity	

The default trace level is 3.

8.3 Error messages with remedial actions