ART internals

Adrian Johnstone

a.johnstone@rhul.ac.uk

August 22, 2025

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

Contents

List of Figures

List of Tables

Preface

This document is a guide to the internal structure of ART and its various algorithms. It is primarily aimed at researchers in programming language specification and implementation, but will also be of interest to those wishing to extend ART or to use it as an API adjunct to their own application code. Readers are expected to already be competent users of ART and to be familiar with the contents the ART user and tutorial guides.

> **Most of the content takes the form of independent notes on aspects of ART. At some point in the future these notes may be smoothed into a coherent single narrative**

ART is a tool for processing textual inputs according to rules that specify the set of acceptable inputs and the effects to be generated by those inputs. We specify the acceptable inputs using *syntax rules* and the corresponding effects using *semantic rules*.

ART is bootstrapped by which we mean that it is self-specified: the ART specification language is specified by a script written in that language, and ART's internal algorithms (lexers, parsers, term rewriters and attribute evaluators) are used to generate ART's effects. Technical details of the bootstrapping mechanism, along with instructions for updating the ART script language will be found in Note 4

Note 1

ART development history

V1 constituted initial experiments in C with the GLL algorithm using core code from the GTB tool.

V2 was the first standalone GLL parser generator, written in C.

V3 began as a port to Java, subsequently massively extended. Most of our research work on parsing has been implemented in V3.

The approach to GLL in V1–3 is as we describe in the papers, that is a parser is written out as a separate piece of code which is then compiled and run. An attribute-action interpreter is included with the parser code which allows L-attributed actions to be executed, in a similar style to RDP, except that the actions are specified in Java, not C.

The core data structure in V1–3 is the representation of a grammar as a tree composed over specialised nodes which correspond to the elements of EBNF grammars such as alternation, sequencing and closures. Derivations are represented by a parallel family of tree node types. These representations turn out to be verbose and rather unweildy.

V4 contains an efficient implementation of immutable terms in the style of CWI's A-terms. Derivations are represented by these terms, and grammars are represented by derivations in the ART scripting language grammar, augmented by maps representing GLL attributes. This approach has proved to be easier to reason about and extend than the V1–V3 style.

V4 also contains a term rewriter, a term based value system and front-end syntax for specifying SOS-style rewrite rules. The value system's implementation is naïve and thus inefficient: it needs to be improved in some future version.

Fora few yars up to AY 2023-24, our UG students have been used V3 to construct attribute-action interpreters, and V4 to construct SOS interpreters. They used V3 to generate the GLL parsers that front-end the SOS interpreters.

V5 provides interpreted implementations of GLL, that is parsers that run in-place without needing to generate a separately compiled parser (although they can do that as well, in both Java and C). The resulting derivations are directly supplied to the rewriter and to the attribute-action interpereter.

In V5, actions are expressions over the value system, not embedded Java or C fragments; the goal being to ensure that interpreted semantics and compiled semantics in Java and in C show the same behaviour. V5 also comes with an integrated development environment (IDE).

Teaching switched to using V5 in AY2024-25 from the command line since the IDE was unfinished.

Note 2

Characters and strings; codepoints and glyphs

Note 3

Context Free Grammars

A context free grammar (CFG) $\Gamma = (N, T, S, P)$ where:
N is a finite set of *nonterminal* symbols,
T is a finite set of *terminal* symbols with $N \cap T = \emptyset$,
$S \in N$ is the *start* nonterminal, and
P is the *produces function* $\rightarrow$ which is a map from N to $(N \cup T)^*$.

Let ϵ denote the empty string.

Let variable X range over N, $\alpha, \beta, \gamma, \delta, \zeta$ range over $(N \cup T)^*$, and u range over T^*

The derivation step relation $\Rightarrow = \{(\alpha, \beta) \mid \alpha = \gamma X \delta, \beta = \gamma \zeta \delta, X \rightarrow \zeta\}$.

The transitive closure of S under $\Rightarrow$ is the set of sentential forms Σ.

$L_\Gamma = \Sigma \cap T^*$ is the set of *sentences* or *language* of Γ.

A derivation of u is a sequence $S \Rightarrow^* u$

A recogniser $R_\Gamma(u)$ yields boolean $\textsc{true}$ iff $u \in L_\Gamma$

A parser $\Pi_\Gamma(u)$ yields the set of all derivations of u in Γ.

Note 4

Bootstrapping, and how to update the ART script language