Translator generation using ART

Adrian Johnstone and Elizabeth Scott[1]

Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
{a.johnstone,e.scott}@rhul.ac.uk

Abstract. ART (Ambiguity Resolved Translators) is a new translator generator tool which provides fast generalised parsing based on an extended GLL algorithm and automatic generation of tree traversers for manipulating abstract syntax. The input grammars to ART comprise modular sets of context free grammar rules, enhanced with regular expressions and annotations that describe disambiguation and tree modification operations using the TIF (Tear-Insert-Fold) formalism. ART generates a GLL parser for the input grammar along with an *output grammar* whose derivation trees are the abstract trees specified by the TIF tree modification operations.

1 Introduction

ART is an integrated generalised parser generator and tree rewriter. ART supports the traditional applications for parser generators (such as compiler front end generation) by providing high performance parsing coupled with parser generation times that are typically less than the time needed to compile the generated code. ART is the first full-scale implementation of both the GLL algorithm [2] and the TIF formalism [3].

Existing generalised parser generators typically use a variant of bottom-up GLR parsing; for example, in ASF+SDF [6], Stratego [1] and Elkhound[4]; even Bison has a partial GLR mode. We have described elsewhere improvements to the GLR algorithm that provide worst-case cubic performance using binarised SPPF's but recognise that users find shift-reduce automata hard to understand. GLL is a generalisation of recursive descent that also provides worst-case cubic performance with linear performance on LL(1) parts of a grammar: in practice we find that GLL runs approximately as fast as our BRNGLR parsers.

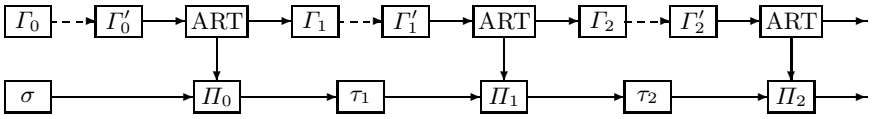
An ART-generated parser or treewalker, Π, performs the following steps (i) lexical analysis of string characters or tree labels into *tokens*; (ii) parsing of those tokens against the grammar to form a *Shared Packed Parse Forest* (SPPF) of derivations; (iii) disambiguation of the potentially many derivations into a single derivation tree; (iv) restructuring of that derivation tree using the TIF operators to form a *restructured derivation tree* (RDT); (v) semantics evaluation.

This paper is mostly concerned with step (iv). The GLL algorithm used in step (ii) was presented at LDTA09 [2] and we shall discuss disambiguation in a future presentation.

On each run, ART produces a parser or treewalker for its input grammar, along with a *TIF Transformed Grammar* (TTG) which completely describes the family of trees that can be constructed by that parser or tree walker. A TTG is itself a TIF grammar, and can thus be used as input to ART, usually after TIF operators specifying the next phase of tree reworking have been added.

The idea is that parsing and tree transformation can be broken down into a series of distinct phases, allowing separation of concerns, but using a single notation and conceptually identical processing at each stage. Our goal is to have a system which is theoretically well founded, but which software engineers and casual users find approachable. The TIF operators are based on the notions of tearing and folding trees using three simple local transforms, coupled with the ability to insert arbitrary tree fragments.

Consider the construction of a C++ to ANSI-C converter such as CFront or of an intermediate language converter such as CIL [5] which simplifies and normalises C++ into a core sub-language in which, for instance, all loops are represented by a single form. We envisage writing such converters as a number of distinct phases, for instance one to handle loop constructs and another to rework data scoping. This separation of concerns between phases allows better testing and in principle reduces the opportunities for unexpected interactions between tree restructurings. We illustrate the process here as a chain of parser/tree walkers $\Pi_0, \Pi_1, \ldots$ with associated TTG's $\Gamma_1, \Gamma_2, \ldots$. At each stage, we envisage the software engineer intervening by adding TIF annotations and by factoring, deleting or multiplying out rules to form a Γ_i'. This process is marked by dashed lines.



The initial inputs are a concrete grammar Γ_0 (say, the ANSI-C standard grammar) and a string σ to be parsed. The translator designer adds TIF annotations to specify the first intermediate form and semantic actions to be executed over trees from that intermediate form. ART then generates a GLL parser/TIF RDT builder Π_0 which processes σ to produce an RDT τ_1. ART also produces the TTG Γ_1 which specifies the full range of τ intermediate forms that can arise from Π_0. Further annotation and reworking of Γ_1 to Γ_1' provides the input to the next stage.

There are significant open questions concerning change management with this approach. As things stand, if a change is made to the grammar is an early phase, the contingent changes to TIF annotations will have to be manually propagated throughout the chain. Ideally we would have some meta-TIF notation that could specify the TIF annotations and thus reapply them automatically. At present

we are attempting to gain experience with the 'manual' approach before trying to abstract a meta-notation.

Part of the motivation for ART comes from our experience of manually constructing simplified equivalent grammars for complex languages such as the C-like assembler for Analog Device's digital signal processor line. Our `asm21toc` reverse compiler started off with a detailed context free grammar (CFG) that captured in the syntax many constraints that arise from the hardware of those processors: for instance shifter operations and arithmetic operations use different sets of input and output registers. We then moved to a grammar which formally accepted a larger language in which, amongst other things, any register could supply the operands for any operator. This *widened* grammar was safe to use because any real program had already been parsed for correctness by the tight grammar: the fact that the grammar could accept programs that were strictly illegal was of no consequence because they had already been filtered out. The widened grammar concentrated operand fetch semantics into one rule, rather than the complicated case analysis that we would have had to implement with the 'real' grammar.

An important feature of our approach in ART is that the tool provides not only a parser, but a formal description of the parser's output which is guaranteed to be complete. A common failure mode when writing systems based on tree rearrangement is to forget about some obscure or infrequently used special case. For instance, a tree-walker based code generator must be able to completely tile any possible intermediate tree with target instructions, and proving that every possible case has been covered is hard if we use *ad hoc* mappings. An engineer working with ART is presented with a grammar which completely describes all possible output trees at each stage of a compiler: by ensuring that every production has appropriate rewrites or semantics, complete coverage is assured.

2 Source syntax, modularity and parsing

The ART source syntax mostly follows the conventions of our earlier RDP and GTB toolkits with added support for modularisation and lexical level rules. We have also created the tools `gramex` and `gramconv` which extract grammar rules from electronically readable standards documents and convert them to the source syntax for a variety of tools including Bison, ASF+SDF, GTB and ART; optionally converting EBNF to BNF using a variety of idioms. These tools and extracted grammars for multiple standardisations of Java, C, C++ and Pascal. are available from

`http://www.cs.rhul.ac.uk/research/languages/projects/grammars/index.html`

An ART specification comprises one or more modules with associated import and export lists. In software engineering, modularisation is used to allow separation of concerns, to encapsulate and abstract away from details, and to support reuse; we believe that the engineering of large grammar-based systems also benefits from effective modularisation. A complete example of an ART specification for a tiny language is shown on page 8.

ART grammar rules use conventional CFG syntax augmented with the postfix regular operators `*`, `+` and `?` for Kleene closure, positive closure and optional items respectively, as well as parentheses and the `|` operator to allow alternates to be gathered together. The symbol `#` represents the empty string ϵ. We also provide the form `A@B` which is shorthand for `A (B A)*`.

To support scannerless parsing, terminals come in three forms. The most fundamental is the *character terminal* denoted by a back-quote followed by either a printable character or one of the ANSI-C conventional character escape sequences. These are designed to be used in lexical level specifications and we also allow the shorthands `‘x-‘z` and `\‘x` to represent sets of character tokens in the range `x` through `z` and the set that includes all character tokens except `x`. The range operator may only be applied between operands that are either both digits, both lowercase or both uppercase characters because ART does not expect any other sequences to have a portable ordering.

Whether ART uses a separate lexer, specified in this way, or deploys the GLL algorithm right down to character level (in the manner of scannerless parsing in SDF [7]) is user selectable: there are theoretical and engineering implications which are beyond the scope of this paper but which will be discussed in future presentations.

Case sensitive terminals are demarcated by single quotes, and represent a shorthand for a whitespace nonterminal followed by the sequence of character terminals corresponding to the pattern of the terminal; that is `’while’` is just shorthand for `(artWhiteSpace ‘w ‘h ‘i ‘l ‘e)`.

The nonterminal `artWhiteSpace` is predefined by ART to correspond to the nonprinting characters; if the user provides explicit productions then the internal default is suppressed. ART specifications are modular, as we describe in the next section, and each module may have its own `artWhiteSpace` definition.

Case insensitive terminals are demarcated by double quotes, and represent a shorthand for a whitespace nonterminal followed by the sequence of character terminals corresponding to the mixed-case pattern of the terminal; that is `"while"` is shorthand for

```
(artWhiteSpace (‘w|‘W) (‘h|‘H) (‘i|‘I) (‘l|‘L) (‘e|‘E)).
```

Nonterminals and terminals may be *named* by appending a colon and an alphanumeric identifier. Names are used in semantic expressions to disambiguate multiple instances of a nonterminal, and to identify torn subtrees that will be reinserted into an RDT.

ART modules export a list of nonterminals. Non-exported nonterminals are private to a module and invisible outside of their parent module. A module import list comprises import entities written as `M.X = Y`. This asks for the productions from module `M` whose left hand side is `X` to be copied into the current module but with their left-hand side name changed to `Y`. The simpler form `M.X` copies the productions for `X` with the same local name as in the exporting module, and the form `M` simply copies all rules that are exported from `M` with their original names. When a rule `X ::= Z1 Z2 ... Zk` is exported by module `M` and imported into module `N` via the instruction `M.X = Y`, the copy rule is properly

written `N.Y ::= M.Z1 M.Z2 ... M.Zk`, in particular if `Zi=X` then `M.Zi` is different from `N.Y`. In a derivation using this rule the grammar rules for `Zi` in module `M` are used to expand `M.Zi`.

Module re-use sometimes demands that we modify imported nonterminal definitions. Extra rules may be added locally simply by writing them into the importing module. However, we can also remove productions from a nonterminal by using the *deleter* `X \ ::=` α. In this case, all rules of the form `X ::=` α (after suppression of TIF annotations and semantic actions) are deleted from the module containing the deleter.

An attempt to recursively import a module is an error, the dependency graph of modules must form a directed acyclic graph. During module resolution, we say a module M is *complete* if all of the modules in its import list are complete, and all imports to M have been performed and all deleters in M have been applied. In practice this means that we start at the leaves of the dependency DAG and work our way back to the root module.

The order of modularity operations for M is as follows:

1. Construct internal representations of all of the rules in the source text for M.
2. When all modules listed as imports to M are complete, execute each import in M and apply renamings.
3. Apply deleters in M to the resulting set of rules.
4. Construct the export list from M.
5. Mark M as complete.

ART uses GLL-style parsing [2] which is a generalisation of recursive descent using the process management regime from our RIGLR parsers. A feature of GLL is that the parser is defined in terms of a small series of *templates* corresponding to various grammar idioms. Converting ART to produce parsers and tree walkers written in a new programming language requires us to write templates in that language and to implement a small set of support routines. ART generates C++; we plan to implement Java templates next.

The present ART implementation uses RDP to generate its front end: ART has not yet been ported to itself. When a bootstrapped version of ART becomes available, some aspects of the source syntax (such as the trailing `;` on productions) will become optional: they are only there to ensure that ART's source syntax is LL(1) and thus admissible by RDP.

A major space component of the generated parsers is the bit strings associated with the selector sets that guard the activation of individual productions. For performance reasons, we do not wish to implement the well known compression techniques used in table driven parsers, but we have observed that for typical programming languages a 75% reduction in space is obtainable simply by storing sets by contents rather than by name because many selector sets have the same contents. For ANSI-C, there are 866 selector sets in the parser but only 218 contents-unique selector sets.

3 Tear-Insert-Fold annotations

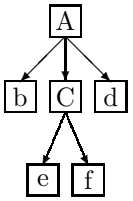
The TIF formalism described in [3] provides four tree manipulation operators. We imagine the tree as being drawn on a piece of paper, and allow for a node to be folded under or over its parent or for complete subtrees to be torn away and then reattached in a different position under the same parent. Essentially we use the derivation tree of a string as a starting point, and then rearrange it into an RDT as specified by the TIF operators.

The four TIF operations are carefully designed to be completely local: we do not allow subtrees to be torn and reattached to any node other than the original parent. This allows us to write an algorithm which generates a specification for the possible output trees in the form of another grammar which we call the TIF Transformed Grammar (TTG): the derivation trees of the TTG are the RDT's of the original grammar.

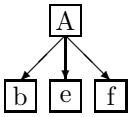
We shall illustrate the TIF operators using the following grammar:

```
A ::= 'b' C 'd';    C ::= 'e' 'f';   X ::= 'y' ;
```

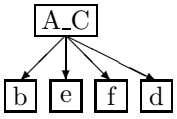
Nonterminal `X` is unreachable from start symbol `A` but will be used in an insertion. The derivation tree for string `befd` is



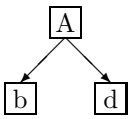
Now, if we add a *fold-under* (`^`) TIF operator to `C` and `d` in `A` giving `A ::= 'b' C^ 'd'^;` the derivation tree is transformed to this RDT:



which has TTG `A ::= 'b' 'e' 'f';` Similarly, if we add the *fold-over* (`^^`) operator giving `A ::= 'b' C^^ 'd';` then the derivation tree would be transformed to this RDT:



with TTG `A_C ::= 'b' 'e' 'f' 'd';`, `A_C` is a new nonterminal. The *tear* operator (`^^^`) removes an entire sub-tree: the rule `A ::= 'b' C^^^ 'd';` yields this RDT:

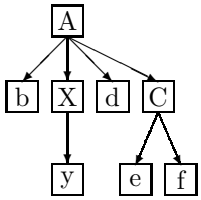


with TTG `A ::= 'b' 'd';` .

We can also perform arbitrary insertions within rules using the `$` operator which may be applied to terminals, nonterminals that generate singleton languages and *named tears*. The TIF production

```
A ::= 'b' C:tearName^^^ [ $X ] 'd' [ $tearName ];
```

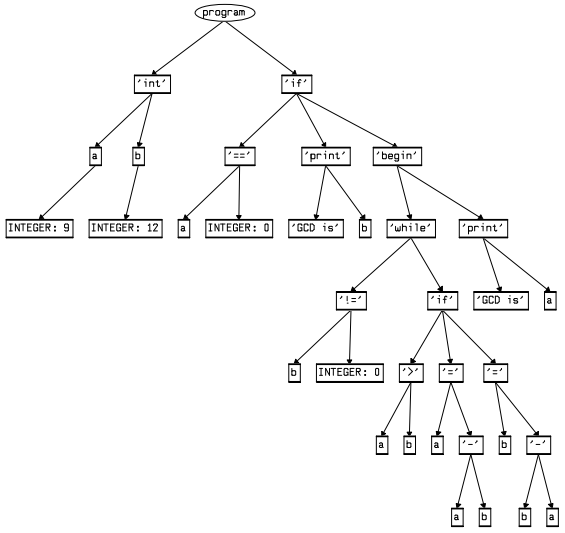
tears the subtree generated b nonterminal C, inserts a tree corresponding to the single derivation in the language of `X` and then re-inserts the torn tree:



with TTG `A ::= 'b' X 'd' C;  C ::= 'e' 'f';  X ::= 'y' ;`

As a more substantial example, here is the source code for Euclid's Greatest Common Divisor algorithm written in the `mini` language originally designed as a tutorial example for RDP together with the RDT used by `mini`'s code generator.

```
int a = 9,
    b = 12;

if a == 0 then
  print("GCD is", b)
else

  begin

    while b != 0 do
      if a>b then
        a = a - b
      else
        b = b - a;

    print("GCD is", a)

  end;
```



The RDT was produced using the following TIF grammar.

```
*mini(declarations statements)(program)
program ::=  #^ | (var_dec | statement | #^ ) ';'^ program^ ;

*declarations(lexer expressions)(var_dec)
var_dec ::= 'int'^^ dec_body  dec_list^ ;
dec_list ::= #^ | ','^ dec_body dec_list^ ;

dec_body ::= id^^ ( '='^ e0 )? ;

*statements(lexer expressions)(statement)
statement ::= id '='^^ e0  |
              'if'^^ e0 'then'^ statement ( 'else'^ statement )? |
              'while'^^ e0 'do'^ statement |
              'print'^^ '('^ ( e0  | stringLiteral ) print_list^ ')'^ |
              'begin'^^ statement stmt_list^ 'end'^;

print_list ::= #^ | ','^ ( e0  | String ) print_list^ ;

stmt_list ::= #^ | ';'^ statement stmt_list^ ;

*expressions(lexer)(e0)
e0 ::= e1 ('>'^^ e1 | '<'^^ e1  | '>='^^ e1 | '<='^^ e1 | '=='^^ e1 | '!='^^ e1)? ;
e1 ::= e2^^ | e1 ('+'^^ | '-'^^) e2 ;
e2 ::= e3^^ | e2 ('*'^^ | '/'^^) e3 ;
e3 ::= e4^^ | '+'^ e3^^ | '-'^^ e3 ;
e4 ::= e5^^ | e5 '**'^^ e4 ;
e5 ::= id^^ | integerLiteral^^ | '('^ e1^^ ')'^ ;

*lexer()(id integerLiteral stringLiteral)
alpha ::= `a..`z | `A..`Z | '_' ;
digit ::= `0..`9;
id ::= alpha (alpha|digit)*;
integerLiteral ::= digit*;
stringLiteral ::= `" (..)* `";
```

4 Some source-to-source conversion examples

The ANSI-C **do-while** and **for** statements can be expressed using the **while** statement. They are provided for user convenience but mapping them onto the **while** statement means that target code has only to be specified for one type of intermediate form. The ANSI-C production describing iteration statements is:

```
iteration_statement ::=
  'while' '(' expression ')' statement  |
  'do' statement 'while' '(' expression ')' ';'  |
  'for' '(' ( expression )? ';' ( expression )? ';'
                                      ( expression )? ')' statement ;
```

The following TIF rules parse the C `for` and `do` constructs respectively but generate RDT's corresponding to the equivalent `while` statements.

```
mappedForLoop ::=
        'for'^ '('^ expr:init^^^ ';'^ expr:test^^^ ';'^ expr:step^^^ ')'^
        [ $init ';' 'while' '(' $test ')' '{' ] statement [ ';' $step '}' ] ';' ;

mappedDoLoop ::=
        'do'^ '{'^ statement:body^^^ '}'^ 'while'^ '('^ expr:test^^^ ')'^ ';'^
        [ $body ';' 'while' '(' $test ')' $body ';' ] ;
```

We can also use TIF annotations to generate a uniform intermediate form from restricted cases of generic constructs. The following TIF rules generate an RDT from a `for` loop which has a root node labelled `for` and exactly four children, three of them valid expression sub-trees and one an instance of statement.

```
expressionFriendlyForLoop ::=
  'for'^^ '(' Eexpr ';' Eexpr';' Eexpr ')' statement ;

Eexpr ::= expression^^ | # ['true'^^] ;
```

Empty expressions are remapped so that if the programmer chooses to omit one of the control expressions we insert the expression `true` instead of leaving a child labelled with the epsilon symbol (or indeed no child at all by some conventions). We choose `true` because the C semantics for `for` specifies that the default for the step expression is that loop execution continues forever. C always discards the result of the initialisation and step expressions; only the side effects are used. Hence any side-effect free expression is a valid default.

5 Concluding remarks and open issues

ART is fast, powerful and unfinished. There are a variety of open issues that we are experimenting with, and we expect to modify the tool's behaviour in response to user experiences.

At present, ART directly implements EBNF parentheses and the `?` operator by multiplying out. Closures are handled by the auxiliary `gramconv` tool. As a result, ART only need generate parser templates for BNF grammars. We have developed parser templates for EBNF constructs which allow iteration within the GLL parser to directly and efficiently handle closures, but the exact form of the trees to be produced is the subject of further study: it is not clear for instance whether a Kleene closure matching the empty string should yield a node labelled ϵ in the SPPF or simply be suppressed.

We have syntax to support lexical level rules, but the exact form of the lexer/parser divide is not specified. ART can produce GLL parsers which truly run at the character level, but the resulting SPPF's can be very large. Alternatively, ART can interface to DFA style lexers.

The fold operators in the TIF formalism are inspired by RDP's fold operators. RDP has been used in a wide variety of industrial and research projects over the last 15 years, and we have confidence that the basic notions of folding are useful and comfortable for engineers. In detail, ART's folds work differently in the case

where we have chains of fold operators, that is when we fold a rule which also has folds on its own right hand side. In RDP, a fold-under operator could promote a fold over operator which then reached up and over the original parent node. We have outlawed this behaviour in ART by ensuring that fold under operators take priority over fold overs. Interestingly, we have never found an instance of this construct in any real RDP grammar.

ART can perform insertions of nonterminals which generate singleton languages, that is languages with only one string. ART builds the derivation tree for that single sentence, and inserts it. In the TIF formalism as described in [3] an insertion may be made of a (N, s) pair in which the derivation of string s in the grammar whose start symbol is N is inserted. ART's present limitation to singleton languages is a restricted version of this: we intend to implement the full semantics in a future version.

Finally, we note the lack of change management capability we need to design a TIF metalanguage which described the annotations to be applied: this could then be interpreted by ART as part of the generation of Γ_i.

References

1. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
2. Adrian Johnstone and Elizabeth Scott. GLL parsing. In *Proc. 9th Workshop on Language Descriptions, Tools and Applications LDTA2009*, 2009.
3. Adrian Johnstone and Elizabeth Scott. Tear-Insert-Fold grammars. *LDTA'10 Tenth Workshop on Language Descriptions, Tools and Applications*, 2010.
4. Scott McPeak and George Necula. Elkhound: a fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *Compiler Construction, 13th Intnl. Conf, CC'04*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004.
5. George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8–12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.
6. M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
7. M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.