UPTR: a simple parse tree representation format

J.J. Vinju
Jurgen.Vinju@cwi.nl

Parsers are important parts of software transformation systems. The surprise is that two different parsers for the same language are notoriously hard to compare on speed or correctness. One of the reasons is that we have no standard representation for the output of parsers.

This abstract describes the Universal Parse Tree Representation exchange format (UPTR). It is a simple, extremely verbose, versatile, efficient, lexer and parser technology independent format. Appendix A contains a full description of the format. I propose to accept this format as a broader standard in the software transformation community, and possibly beyond.

UPTR is *not* new. It is an adaptation of AsFix, which is an existing parse tree exchange and manipulation format introduced by the Meta-Environment [3]. This format was simplified by Visser [9]. UPTR removes some ideosyncrasies, adds character encoding as a parameter, and adds a representation for cyclic derivations.

AsFix has been in use by other software systems since 1999, including StrategoXT [10], and Elan [2]. It is used as an exchange format between front-ends and back-ends of software transformation systems, as well as a back-to-back vehicle for source code transformations. It is extensively used in both batch tools and interactive tools for source code transformation. AsFix is applied in both academic and industrial contexts.

UPTR avoids abstraction. A *parse tree* (PT) is an exact representation of a specific derivation tree[1] of a specific sentence for a specific context-free grammar (CFG) for a specific language (L). A *parse forest* (PF) is a collection of alternative *parse trees* for the same sentence using the same CFG. A PT is by definition acyclic, a PF need not be acyclic[2]. An *abstract syntax tree* (AST) is any arbitrary tree-shaped abstraction of a parse tree.

Any abstraction from a PT may result in the loss of valuable information (e.g. source code comments) [8]. The original source code may not be reconstructable from an AST. Having no abstraction at all guarantees that there is less implicit semantics hidden in the output of a parser. A PT is just a derivation, nothing more and nothing less. This is the foremost property of UPTR.

[1] A proper definition of a derivation tree can be found in Sudkamp [7] on page 49.
[2] A cyclic derivation reaches a non-terminal twice without consuming any terminal symbols.

UPTR supports the construction of the entire derivation, including lexical structure [9, 8]. It can represent character classes and regular expressions, so anything lexical analysis produces may be represented.

UPTR supports the construction of parse forests. It can effectively represent many derivations in a single data structure. All forms of generalized parsers that produce several derivations are therefore serviced. The use of the forest constructor in UPTR does not enforce a minimal forest representation, it can be used to encode any choice point in a derivation tree. Even cyclic derivations may be represented using UPTR.

UPTR is efficient despite its verbosity. UPTR is intentionally verbose. A print-out of a parse tree of the string `true and false` over a grammar for Booleans will pretty much fill your screen. The verbosity makes the format simple and informative, but is this an efficient enough representation? Not without some tricks.

The syntax of the format (Appendix A) strictly follows the ATerm syntax [4]. Therefore, any parse tree is an ATerm. ATerms are an efficient data exchange format with a choice for either efficient or readable serialization. ATerms are implemented using maximal sharing. This annihilates most of the overhead of the verbosity of our format, since every unique `Prod`, `Sym`, or `Tree` is represented at a single memory location. ATerms are used much more widely than only for parse trees [5]. Using ATerms as the implementation vehicle, the parse tree format can be both verbose and efficient. Another benefit of the ATerm library is that it comes with automatic garbage collection in C.

The efficiency of programming remains to be discussed. A functional-style API is provided in C and Java to hide to ATerm implementation layer. Furthermore, if you have a fixed CFG, a code generator may be used to hide the UPTR layer behind a language specific API [6]. The resulting effect is an AST-like interface hiding a PT representation.

Apart from verbosity, XML does not offer any of the features that ATerms offer to make UPTR trees efficient. If you insist, any ATerm may be mapped to an XML document. For now, there is evidence that XML will not suffice for representing source code [1]. More analysis of the relationship between ATerms and XML can be found in [5].

UPTR is versatile. It offers three extension points for arbitrary markup of the information it contains. Firstly, any *production* may have an arbitrary number of attributes of arbitrary shape attached to it. Secondly, every *tree* may have an arbitrary number of annotations. Finally, the list of *symbols* is in principle open-ended. These extension points make it possible to store enough information in a PT such that ample communication is possible between a parser front-end and a particular back-end. Finally, UPTR is parameterized by the character encoding.

Conclusion. The UPTR proposal would be a step towards robust comparison, validation and more reuse of parsers. We would be able to analyze the output of a parser without first reverse engineering the formalism in which the structures are expressed.

It would allow us fixate both parser input and parser output. So, comparing the speed of parser algorithms and implementations would become more honest and more precise.

Each tree would contain the grammar that was used to construct it. So, comparing the actual structure of parse trees could be done without first reverse engineering the grammar. This would also allows users to compare parsers independent of the way they are constructed (generated).

Appendix A. Annotated BNF of the parse forest format. [3] [4] [5]

`Tree`	`::=`	`Integer`	*A leaf node with a character value.*
	`|`	`appl(Prod,Args)`	*One node in a derivation.*
	`|`	`amb(Args)`	*Choice of derivations.*
	`|`	`cycle(Sym,Integer)`	*A cut link of a cyclic derivation.*
	`|`	`Tree "{" Annos "}"`	*Every tree can be annotated.*
`Args`	`::=`	`"[" {Tree ","}* "]"`	*List of trees.*
`Prod`	`::=`	`prod(Syms,Sym,Attrs)`	*Representation of one alternative.*
	`|`	`list(Symbol)`	*Abstraction for list productions.*
`Syms`	`::=`	`"[" {Sym ","}* "]"`	*List of symbols*
`Sym`	`::=`	`"empty"`	*The empty terminal symbol*
	`|`	`sort(String)`	*A non-terminal name.*
	`|`	`lit(String)`	*Literal (terminal).*
	`|`	`seq(Syms)`	*Sequence constructor.*
	`|`	`opt(Sym)`	*Optional constructor.*
	`|`	`alt(Syms)`	*Alternative constructor.*
	`|`	`iter-plus(Sym)`	*Non-empty list constructor.*
	`|`	`iter-star(Sym)`	*List constructor.*
	`|`	`iter-plus-sep(Sym,Sym)`	*Non-empty separated list constructor.*
	`|`	`iter-star-sep(Sym,Sym)`	*Separated list constructor.*
	`|`	`var(Sym)`	*Meta variables.*
	`|`	`char-class(Code,Ranges)`	*Character classes*
	`|`	`...`	*Extensible here!*
`Code`	`::=`	`String`	*Name of encoding table ("ASCII")*
`Ranges`	`::=`	`"[" {Range ","}* "]"`	*List of ranges.*
`Range`	`::=`	`Integer`	*Single character value.*
	`|`	`range(Integer,Integer)`	*Inclusive range of characters.*
`Attrs`	`::=`	`"[" {Attr ","}* "]"`	*List of production attributes.*
`Attr`	`::=`	`ATerm`	*Any ATerm is an attribute*
`Annos`	`::=`	`"[" {Anno ","}* "]"`	*List of tree annotations.*
`Anno`	`::=`	`"["ATerm "," ATerm"]"`	*ATerm key/value pair.*

[3] Notation: "`foo(Bar,Rab)`" abbreviates "`"foo" "(" Bar "," Rab ")"`".
[4] Notation: "`{A B}*`" is a list of A's *separated* by B's.
[5] Notation: "`Integer`", "`String`", and "`ATerm`" are not defined here but fixed anyway.

References

[1] P. Anderson. The performance penalty of XML for program intermediate representations. In J. Krinke and G. Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, September 2005.

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *WRLA*, volume 15 of *ENTCS*. Elsevier Sciences, 1998.

[3] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[4] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[5] M.G.J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: it's all about sharing! In *Special issue on information and software technology*. Elsevier, 2006. To appear.

[6] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, 2004.

[7] T.A. Sudkamp. *Languages and Machines an Introduction to the Theory of Computer Science*. Addison Wesley, June 1994.

[8] J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, Universiteit van Amsterdam, November 2005.

[9] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[10] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.