Annotated Parse Trees
for a Language Parametric IDE
Experience Report from The Meta-Environment

Jurgen Vinju

CWI, Amsterdam

October 22th, 2007
Overview

1. Introduction
2. Global architecture
3. Low level design
4. Pros and cons
5. References and discussion
The ASF+SDF Meta-Environment

- What is it?
  - 1990's: Generated/generic IDE based on formal specification
  - 2000's: Generic meta tool construction platform
  - *Generic* means *any* language and *any* transformation or analysis
The ASF+SDF Meta-Environment

What is it?
- 1990’s: Generated/generic IDE based on formal specification
- 2000’s: Generic meta tool construction platform
- Generic means any language and any transformation or analysis

Application areas:
- Compiler and IDE construction for DSL’s
- Source-to-source transformations for legacy languages
- Large scale source code analysis and transformation
The **ASF+SDF Meta-Environment**

- **What is it?**
  - 1990’s: Generated/generic IDE based on formal specification
  - 2000's: Generic meta tool construction platform
  - *Generic* means *any* language and *any* transformation or analysis

- **Application areas:**
  - Compiler and IDE construction for DSL’s
  - Source-to-source transformations for legacy languages
  - Large scale source code analysis and transformation

- **Architecture:**
  - Started as homogeneous, functional, event-based, incremental
  - Reimplemented as heterogeneous, modular, message-based, open
Composition and coordination from the outside

![Diagram of architecture and components](image-url)

**Key**
- = component
- = component with multiple instances
- = ToolBus
- = interacts with
Language centric design

grammars as contracts & parse trees as ubiquitous media
Implementation of parse trees

- Parse Trees:
  - Each node is an application of a grammar rule
  - Each node has as much children as the grammar rule has members
  - Leaves are the characters of the source code
  - Nodes also encode structure of lexical syntax
  - Each node can have an arbitrary amount of annotations
Implementation of parse trees

- **Parse Trees:**
  - Each node is an application of a grammar rule
  - Each node has as much children as the grammar rule has members
  - Leaves are the characters of the source code
  - Nodes also encode structure of lexical syntax
  - Each node can have an arbitrary amount of annotations

- **In memory and serialized representation as “ATerms”:**
  - Maximal sub-term sharing (via hash-consing)
  - Equality in $O(1)$
  - Efficient (de-)serialization (textual/shared/binary)
  - Implementations in C, Java, . . .
  - Garbage collection in C.
Little example grammar

LAYOUT ::= [\n\t\n ]
BoolCon ::= "true"
         | "false"
Boolean ::= BoolCon
         | Boolean "&" Boolean {left}

- Little parse trees have huge representations
- Sharing helps a little for little parse trees
- Sharing helps a lot for larger parse trees
Textual parse tree for “true & false” (1824 bytes)

```
[jurgenv-laptop,jurgenv] ~/drafts/plide2007 > echo "true & false" | sglr -p bool.tbl | trmcat
parsetree(appl(prod([cf(opt(layout))], cf(sort("Boolean")), cf(opt(layout))), sort("<START>")], no-attrs), [appl(prod([cf(layout)], cf(opt(layout))], no-attrs), [appl(prod([lex(iter(layout))], cf(layout))), no-attrs], [appl(prod([cf(sort("Boolean"))], cf(sort("Boolean"))), attr([assoc(left), term(cons("and"))])), [appl(prod([cf(sort("BoolCon"))], cf(sort("Boolean"))), attr([term(cons("constant"))))], [appl(prod([lit("true"), cf(sort("BoolCon"))], attr([term(cons("true"))))], [appl(prod([char-class([116]), char-class([114]), char-class([117]), char-class([101])]), lit("true"), no-attrs], [116, 114, 117, 101)]))))], [appl(prod([cf(layout)], cf(opt(layout))], no-attrs), [appl(prod([lex(iter(layout))], cf(layout))), no-attrs], [appl(prod([lex(iter(layout))], [appl(prod([char-class([range(9, 10), 13, 32)])], lex(layout), attr([term(cons("whitespace")), id("basic/Booleans"))], [32]))])], [appl(prod([char-class([38])], lit("&"), no-attrs], [38]), appl(prod([cf(layout)], cf(opt(layout))], no-attrs), [appl(prod([lex(iter(layout))], cf(layout), no-attrs), [appl(prod([char-class([range(9, 10), 13, 32)])], lex(layout), attr([term(cons("whitespace")), id("basic/Booleans"))], [32]))]), appl(prod([cf(sort("BoolCon"))], cf(sort("Boolean"))), attr([term(cons("constant"))))], [appl(prod([lit("false"), cf(sort("BoolCon"))], attr([term(cons("false"))))], [appl(prod([char-class([102]), char-class([97]), char-class([108]), char-class([115]), char-class([101])]), lit("false"), no-attrs], [102, 97, 108, 115, 101)])]), appl(prod([cf(layout)], cf(opt(layout))], no-attrs), [appl(prod([lex(iter(layout))], cf(layout), no-attrs), [appl(prod([char-class([range(9, 10), 13, 32)])], lex(layout), attr([term(cons("whitespace")), id("basic/Booleans"))], [10]))]))], 0)
```
Shared parse tree for “true & false” (635 bytes)

```
tbl | baffle -ws
!parsetree(appl(prod([cf(opt(layout)),cf(sort("Boolean")),#C],sort("<START>"),no-
-attrs),[appl(prod([cf(#A)],#C,#J),[appl(prod([lex(iter(#A))],#L,#J),[appl(list( #P),[])]),appl(prod([#F,#C,lit("&"),#C,#F],[F],attrs[[assoc(left),term(cons("an-
d"))]],),[appl(prod([cf(sort("BoolCon")),#F],attrs[[term(cons(cons("constant":)))])),[appl( prod([lit("true"),#l,attrs[[term(cons(#t))]]),[appl(prod([char-class([116]),
char-class([114]),char-class([117]),char-class([101]),#u,#J],[#1,#4,#7,#+)]),
),appl(#N,[appl(#R,[appl(#S,[appl(prod([char-class([range(9,10),13,32])],lex(#A),
attrs[[term(cons("whitespace")),id("basic/Booleans")]),[32]])],),appl(prod([char-
class([38])],[Z,#J],#Be),#Bd,appl(#s,[appl(prod([lit("false")],#l,attrs[[term( cons(#Bj)])]),),appl(prod([char-class([102]),char-class([97]),char-class([108])
],char-class([115]),#BA],[Bk,#J),[102,97,108,115,#+)]),][)),appl(#N,[appl(#R,[appl( #S,[appl(#BV,[10])])])))],0)]
jurgenv-laptop,jurgenv] ~/drafts/plide2007 >
```
Binary parse tree for “true & false” (583 bytes)

```
tbl | baffle -WS
>D parsetree appl prod cf opt layoutA sort!Boolean A !<START
>no-attrs A A A    
    A A  lex iter    
        A list A A  lit!
& attr assoc left term cons! and A A A A !BoolCon A A A constant
! true, A A A !9 A A
char-class tA rA uA e8
   D G J O MA A A A A range
    A
  A A A A!
whitespace id! " " /B/ /r/ $ A A A &€ €
! "/r/ €, A A A t A A A fA aA lA sKs €* A A A
 A A A
```
**Pros**

- Parse tree processors are simple functional programs
- Readable and debuggable
- No a priori abstraction at all
- Space efficient, exploiting natural redundancy
- Grammars as contracts, API generation
- Tools written in different programming languages
- Source code at your fingertips
- Grammar at your fingertips
**Pros**
- Parse tree processors are simple functional programs
- Readable and debuggable
- No a priori abstraction at all
- Space efficient, exploiting natural redundancy
- Grammars as contracts, API generation
- Tools written in different programming languages
- Source code at your fingertips
- Grammar at your fingertips

**Cons**
- Sharing is hampered by annotations
- Equality modulo whitespace is in $O(n)$
- No destructive updates in trees, implies functional style of programming
- Requires different optimization strategies
- Need high-end efficient serialization
- Need to think about all contracts a priori
References

- *The ASF+SDF Meta-Environment*
  http://www.meta-environment.org

- *Toward an engineering discipline for grammarware*
  Lämmel, Klint, Verhoef

- *Grammars as Contracts*
  De Jonge, Visser

- *Efficient Annotated Terms*
  Van den Brand, Olivier, De Jong

- *Generation of abstract programming interfaces from syntax definitions*
  Olivier, De Jong

- *Generator of efficient strongly typed abstract syntax trees in java*
  Vinju, Moreau, Van den Brand
Forums

- Language Descriptions, Tools and Applications (LDTA)
  Deadline December 7th
- Workshop on Software Transformation Systems (STS)
- Generative Programming and Component Engineering (GPCE)
- SCP special issues on Experimental Software and Toolkits (EST)
- SPE
- ...
Discussion

- Abstraction considered harmful. “Syntax Trees” as opposed to “Abstract Syntax Trees”.
- What are the minimal requirements on AST hierarchies for IDE construction?
- Maximal sub-term sharing: friend or foe?
- Grammars as contracts: formalized serialization interfaces imply open extensibility?