Abstract Syntax Sucks!

deconstruction (?), allegory (?), ...

Tijs van der Storm
Alfred Aho
(contributed to lex)
Alfred Aho
(contributed to lex)

Scanners suck!

Jurgen Vinju
“Deconstruction”

• Turn hierarchies up-side-down
• Bring margins to the center
Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY

In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.
From the original notes...

Basic irrelevance of syntax and primacy of semantics.

http://fexpr.blogspot.nl/2011/06/primacy-of-syntax.html
But syntax is the UI!?! 

And syntax is part of that, NOT all of it, but nonetheless a very visible aspect of it.
Topsy turvy

- Concrete syntax over abstract syntax
- Implementation/engineering over semantics
- Comments, whitespace, layout, etc.
A compiler pipeline...

Parse → "Implode" → Parse tree → AST → The rest

```java
public X getX(int i) {
    return rows.get(i);
}
public X getY(int i) {
    return cols.get(i);
}
public int lastIndexX() {
    return rows.size() - 1;
}
public int lastIndexY() {
    return cols.size() - 1;
}
```
public X getX(int i) {
    return rows.get(i);
}

public X getY(int i) {
    return cols.get(i);
}

public int lastIndexX() {
    return rows.size() - 1;
}

public int lastIndexY() {
    return cols.size() - 1;
}

Parse

Parse tree

“Implode”

AST

The rest
This talk

Parse

“Implode”

AST

The rest
Rascal Language Workbench

- “Functional” meta-programming language
- DSL implementation and program understanding/renovation
- Source code in, source code out
- Built-in context-free grammars
- Pattern matching, traversal, comprehensions, relation algebra, ...

http://www.rascal-mpl.org
Language Workbench Challenge

```java
def taxOfficeExample {
    "Did you sell a house in 2010?"
    boolean hasSoldHouse
    "Did you buy a house in 2010?"
    boolean hasBoughtHouse
    "Did you enter a loan?"
    boolean hasMaintLoan
    if (hasSoldHouse) {
        "What was the selling price?"
        money sellingPrice
        "Private debts for the sold house:"
        money privateDebt
        "Value residue:"
        money valueResidue = (sellingPrice - privateDebt)
    }
}
```
The State of the Art in Language Workbenches

Conclusions from the Language Workbench Challenge

Sebastian Erdweg\textsuperscript{1}, Tijs van der Storm\textsuperscript{2,3}, Markus Völter\textsuperscript{4}, Meinte Boersma\textsuperscript{5}, Remi Bosman\textsuperscript{6}, William R. Cook\textsuperscript{7}, Albert Gerritsen\textsuperscript{6}, Angelo Hulshout\textsuperscript{8}, Steven Kelly\textsuperscript{9}, Alex Loh\textsuperscript{7}, Gabriël Konat\textsuperscript{10}, Pedro J. Molina\textsuperscript{11}, Martin Palatnik\textsuperscript{6}, Risto Pohjonen\textsuperscript{9}, Eugen Schindler\textsuperscript{6}, Klemens Schindler\textsuperscript{6}, Riccardo Solmi\textsuperscript{12}, Vlad Vergu\textsuperscript{10}, Eelco Visser\textsuperscript{10}, Kevin van der Vlist\textsuperscript{13}, Guido Wachsmuth\textsuperscript{10}, and Jimi van der Woning\textsuperscript{13}

\textsuperscript{1}TU Darmstadt, Germany \textsuperscript{2}CWI, Amsterdam, The Netherlands \textsuperscript{3}INRIA Lille Nord Europe, Lille, France \textsuperscript{4}voelter.de, Stuttgart, Germany \textsuperscript{5}DSL Consultancy, Leiden, The Netherlands \textsuperscript{6}Sioux, Eindhoven, The Netherlands \textsuperscript{7}University of Texas, Austin, US \textsuperscript{8}Delphino Consultancy, Best, The Netherlands \textsuperscript{9}MetaCase, Jyväskylä, Finland \textsuperscript{10}TU Delft, The Netherlands \textsuperscript{11}Icinetic, Sevilla, Spain \textsuperscript{12}Independent, Bologna, Italy \textsuperscript{13}Universiteit van Amsterdam

Abstract.

Language workbenches are tools that provide high-level mechanisms for the implementation of (domain-specific) languages. Language workbenches are an active area of research that also receives many contributions from industry. To compare and discuss existing language workbenches, the annual Language Workbench Challenge was launched in 2011. Each year, participants are challenged to realize a given domain-specific language with their workbenches as a basis for discussion and comparison. In this paper, we describe the state of the art of language workbenches as observed in the previous editions of the Language Workbench Challenge. In particular, we capture the design space of language workbenches in a feature model and show where in this design space the participants of the 2013 Language Workbench Challenge reside. We compare these workbenches based on a DSL for questionnaires that was realized in all workbenches.

1 Introduction

Language workbenches, a term popularized by Martin Fowler in 2005 \cite{fowler2005}, are tools that support the efficient definition, reuse and composition of languages and their IDEs. Language workbenches make the development of new languages affordable and, therefore, support a new quality of language engineering, where sets of syntactically and semantically integrated languages can be built with comparably little effort. This can lead to multi-paradigm and language-oriented programming environments \cite{gunther2009, larsen2003} that can address important software engineering challenges.

Almost as long as programmers have built languages, they have also built tools to make language development easier and language use more productive. The earliest language workbench probably was SEM \cite{seemann1972}; other early ones include MetaPlex \cite{allen1994}, Metaview \cite{cooper1996}, QuickSpec \cite{goble1994}, and MetaEdit \cite{ruiz1995}. Graphical workbenches that are still being developed today include MetaEdit+ \cite{meeden2006}, DOME \cite{ruiz1995}, and GME \cite{schott2013}. On
Rascal’s concrete syntax feature

• Grammar non-terminals are types
• Parse trees are values
• Concrete syntax pattern matching and construction
• All parse trees subtype of Tree
  • (an ADT for parse trees)
module Syntax
extend lang::std::Layout;

start syntax Controller =
  controller:
    Events events
    ResetEvents? resets
    Commands? commands
    State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
  controller:
      Events events
      ResetEvents? resets
      Commands? commands
      State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
module Syntax
extend lang::std::Layout;

start syntax Controller =
  controller:
    Events events
    ResetEvents? resets
    Commands? commands
    State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
Syntax definition

```plaintext
module Syntax
extend lang::std::Layout;

start syntax Controller =
    controller:
    Events events
    ResetEvents? resets
    Commands? commands
    State+ states;

syntax Events
    = "events" Event* "end";

syntax ResetEvents
    = "resetEvents" Id* "end";

syntax Commands
    = "commands" Command* "end";
```
module Syntax
extend lang::std::Layout;

start syntax Controller =
controller:
  Events events
ResetEvents? resets
Commands? commands
State+ states;

syntax Events
  = "events" Event* "end";
syntax ResetEvents
  = "resetEvents" Id* "end";
syntax Commands
  = "commands" Command* "end";
lexical Id
    = ([a-zA-Z][a-zA-Z0-9_]*)!>>[a-zA-Z0-9_])
\ Reserved ;

keyword Reserved
    = "events"
    | "end"
    | "resetEvents"
    | "state"
    | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]*) !>> [a-zA-Z0-9_])
  \ Reserved ;

keyword Reserved
  = "events"
   | "end"
   | "resetEvents"
   | "state"
   | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
  \ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]*) \> [a-zA-Z0-9_])
  \ Reserved ;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions" ;

lexicals don’t get layout

follow restriction

character class
Lexical syntax

lexical Id
  = ([a-zA-Z][a-zA-Z0-9_]* !>> [a-zA-Z0-9_])
\ Reserved;

keyword Reserved
  = "events"
  | "end"
  | "resetEvents"
  | "state"
  | "actions";
Lexical syntax

lexical Id
= ([a-zA-Z][a-zA-Z0-9_]*) !>> [a-zA-Z0-9_])
\ Reserved;

keyword Reserved
= "events"
| "end"
| "resetEvents"
| "state"
| "actions" ;
Parse trees
Demo

• 1: parse trees in Rascal
• 2: concrete matching and construction
• 3: “analyzing” comments
WYSIWYG

- Concrete: \( x + y \)
- Abstract: `add(var("x"), var("y"))`
“Pretty” printing for free

- With ASTs: printing = pretty printing
- AKA: inventing layout
- Parse trees can be *unparsed*
- Text can be highlighted based on parse trees
High-fidelity transformation

- Need to preserve comments/layout for
  - Refactoring
  - Renovation
if(hasSoldHouse) {
    /*
    * We only ask for <sellingPrice> and <privateDebt>
    * if <hasSoldHouse == true>
    */
    "What was the selling price of the house?"
    money sellingPrice
    "Private debts for the sold house:"
    money privateDebt
    "Value residue:"
    money valueResidue = (sellingPrice - privateDebt)
}
if(hasSoldHouse) {
    /*
    * We only ask for <sellingPrice> and <privateDebt> if <hasSoldHouse> > 1
    */
    "What was the selling price of the house?"
    money sellingPrice
    "Private debts for the sold house:"
    money privateDebt
    "Value residue:"
    money valueResidue = (sellingPrice - privateDebt)
}
Summary

- ASTs: discard layout/comments
- Parse trees: contain all of it
  - high-fidelity transformation
  - “comments are part of the language too”
- Rascal:
  - parse trees, concrete matching, Tree
Conclusion

• Abstract syntax sucks ;)

• Anything you can do abstract, I can do concrete...

• ...and more.

• Abstraction is great, but has its cost.