Imperative and Declarative Views on Software Language Evolution

Vadim Zaytsev
Universiteit van Amsterdam
NLFP 2014
Introduction

- Universiteit van Amsterdam (2013–2014)
- Vrije Universiteit Amsterdam (2004–2008)

Vadim Zaytsev
Introduction

- Rascal (2010–2013)
- Prolog (2008–2010)
- Smalltalk (2004–2008)

Vadim Zaytsev
Part I
SLE background
Software Languages

Programming languages
### Software Languages

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Functional languages</th>
</tr>
</thead>
</table>

---

**1977 ACM Turing Award Lecture**

The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jane E. Sammon, Chairman of the Awards Committee, made the following comments and read a portion of the full citation. The full announcement is in the September 1977 issue of Communications, page 881.

"Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don’t really know what they stand for. Well, the B is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus, both in the Fortran case and in the contributions that he is receiving this year’s Turing award.

The full citation is very profuse, influential, and lasting contributions to the design of practical high-level programming systems; notably through his work on Fortran, and for seminal publications of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows:

... Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group’s efforts was a high-level language for scientific and technical computation called Fortran. This same group designed the full system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. national standard in 1964.

During the latter part of the 1950s, Backus served on the international committee which developed Algol 60 and a later version, Algol 68. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1958, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In the paper, he was the first to define and describe a formal system for the specification of programming languages. The formal notation became known as ‘BNF’—standing for ‘Backus Normal Form’, or ‘Backus Notation’ to recognize the further contributions by Per Enner of Denmark. Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational languages. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition."

---

**Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs**

John Backus

IBM Research Laboratory, San Jose

Conventional programming languages are growing ever more numerous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structural data, are often nonprocedural and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Communications of the ACM
August 1978
Volume 21
Number 8

© 1978 ACM. 0001-0782/78/0800-0613 $00.75

Author’s address: 91 Santa Gertrudis Ave., San Francisco, CA 94114.

© 1978 ACM. 0001-0782/78/0800-0613 $00.75
Software Languages

- Programming languages
- Functional languages
- Declarative languages
Software Languages

- Programming languages
- Functional languages
- Declarative languages
- Modelling languages
Software Languages

- Programming languages
- Functional languages
- Declarative languages
- Modelling languages
- Markup languages
- ...

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>XYZ</title></head>
  <body><p>voluptatem accusantium doloremque</p></body>
</html>
```

http://commons.wikimedia.org/wiki/File:XHTML.svg
Software Language Evolution

- Language $\rightarrow$ next version
  - more features
  - backward compatibility
- DSL $\rightarrow$ DSL
  - typically developed iteratively
  - feedback from client, performance, etc
Software Language Evolution

- Language → language dialect
  - some features added, others blocked
  - possibly concrete syntax deviation
- Language description → technology-specific one
  - esp. parsing techniques
- Language → language replication
  - compatibility
Grammar (in a broad sense)

- Definition of a software language
- Commitment to structure
- Differentiates between ‘correct’ and ‘incorrect’
- Comes in various flavours
  - parser specs, metamodels, class diagrams,
    (G)ADTs, XML schemata, ontologies, protocols,
    APIs, documentation, ...
- A finite definition of a (possibly) infinite language
Grammar (in a broad sense)

- Nonterminals (syntactic categories)
- Terminals (atomic symbols)
- Labels, markers, groups
- Repetitions (?, +, *, seplists)
- Disjunction (conjunction, negation)
- ...
- Equivalence problem is undecidable
Grammar example (ADT)

```haskell
{-# OPTIONS -fglasgow-exts #-}
module Types where

import Data.Generics

data Function = Function Name [Name] Expr
  deriving (Eq,Show,Typeable,Data)

type Name = String

data Expr =
  Literal Int
  | Argument Name
  | Binary Ops Expr Expr
  | IfThenElse Expr Expr Expr
  | Apply Name [Expr]
  deriving (Eq,Show,Typeable,Data)

data Ops = Equal | Plus | Minus
  deriving (Eq,Show,Typeable,Data)
```
Function ::= [Function]::(Name Name* Expr);
Name ::= String;
Expr ::= [Literal]::Int
| [Argument]::Name
| [Binary]::(Ops Epr Expr)
| [IfThenElse]::(Expr Expr Expr)
| [Apply]::(Name Expr*);
Ops ::= [Equal]::ε
| [Plus]::ε
| [Minus]::ε;
Part II
Imperative View
Imperative view on software language evolution

Grammar 1 ➔ Grammar 2
Imperative example
Imperative example
Imperative example
Imperative example

Grammar differences

- intended vs. accidental
- result of grammar adaptation
- result of grammar evolution
- idiosyncrasies thanks to metanotation
- idiosyncrasies thanks to parsing technology
- presentation and understandability
- misspelling
- ...etc
Part III
Declarative View
Declarative view on software language evolution
Declarative view on software language evolution

Transformation

Input G
Declarative view on software language evolution
Declarative example

expr : ...;
atom : ID | INT | (' expr ');

abstractize

expr : ...;
atom : ID | INT | expr;

vertical

expr : ...;
atom : ID;
atom : INT;
atom : expr;

unite

expr : ...;
expr : ID;
expr : INT;

abridge

expr : ...;
expr : expr;
Table 7: XBGF operators usage for JLS convergence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>jls1</th>
<th>jls12</th>
<th>jls123</th>
<th>jls2</th>
<th>jls3</th>
<th>read12</th>
<th>read123</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>10</td>
<td>—</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>reroott</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>unfold</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>2</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>fold</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>11</td>
<td>13</td>
<td>2</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>inline</td>
<td>3</td>
<td>67</td>
<td>8</td>
<td>71</td>
<td>100</td>
<td>—</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>extract</td>
<td>—</td>
<td>17</td>
<td>5</td>
<td>18</td>
<td>30</td>
<td>—</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>chian</td>
<td>1</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>massage</td>
<td>2</td>
<td>13</td>
<td>—</td>
<td>15</td>
<td>32</td>
<td>5</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>distribute</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>18</td>
</tr>
<tr>
<td>factor</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>24</td>
<td>3</td>
<td>1</td>
<td>44</td>
</tr>
<tr>
<td>deyaccify</td>
<td>2</td>
<td>20</td>
<td>—</td>
<td>25</td>
<td>33</td>
<td>4</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>yaccify</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>2</td>
</tr>
<tr>
<td>eliminate</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>14</td>
<td>22</td>
<td>—</td>
<td>—</td>
<td>46</td>
</tr>
<tr>
<td>introduce</td>
<td>—</td>
<td>1</td>
<td>30</td>
<td>4</td>
<td>13</td>
<td>3</td>
<td>34</td>
<td>85</td>
</tr>
<tr>
<td>import</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>vertical</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>22</td>
<td>5</td>
<td>8</td>
<td>62</td>
</tr>
<tr>
<td>horizontal</td>
<td>4</td>
<td>19</td>
<td>5</td>
<td>17</td>
<td>31</td>
<td>4</td>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>14</td>
<td>13</td>
<td>7</td>
<td>20</td>
<td>28</td>
<td>20</td>
<td>103</td>
</tr>
<tr>
<td>appear</td>
<td>—</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>25</td>
<td>2</td>
<td>17</td>
<td>71</td>
</tr>
<tr>
<td>widen</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>upgrade</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>14</td>
<td>20</td>
<td>2</td>
<td>2</td>
<td>46</td>
</tr>
<tr>
<td>unite</td>
<td>18</td>
<td>2</td>
<td>—</td>
<td>18</td>
<td>21</td>
<td>5</td>
<td>4</td>
<td>68</td>
</tr>
<tr>
<td>remove</td>
<td>—</td>
<td>10</td>
<td>1</td>
<td>11</td>
<td>18</td>
<td>—</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>disappear</td>
<td>—</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>—</td>
<td>—</td>
<td>33</td>
</tr>
<tr>
<td>narrow</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>4</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>downgrade</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>8</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>13</td>
</tr>
<tr>
<td>define</td>
<td>—</td>
<td>6</td>
<td>—</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>undefined</td>
<td>—</td>
<td>3</td>
<td>—</td>
<td>5</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>11</td>
</tr>
<tr>
<td>redefine</td>
<td>—</td>
<td>3</td>
<td>—</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>inject</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>4</td>
<td>—</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>project</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>4</td>
</tr>
<tr>
<td>replace</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>unlabel</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Grammar mutations

distribute \(\vdash\) DistributeAll
eliminate \(\vdash\) EliminateTop
concatT \(\vdash\) ConcatAllT
inline \(\vdash\) InlineLazy
renameN \(\vdash\) RenameNUpperDash2CamelNone
define \(\vdash\) DefineAll([pi])
Part IV
Imperative vs Declarative
Imperative View on Evolution

- Easy to use
  - no extra effort required
  - no additional languages involved
- No intention tracked
  - what actually changed?
  - what changed conceptually?
  - why was it changed?
Declarative View on Evolution

- Hard to use
- Tedious to specify each change
- Need to learn/develop a new language

Transformations are first class entities

- Can be saved, documented, reused, rerun
- Can be inspected without execution
- Can be transformed on its own
Both approaches have (dis)advantages

Declarative $\rightarrow$ imperative
- easy, if the input is given

Imperative $\rightarrow$ declarative
- need a special ‘grammar differ’
Equality-based differ

- Equivalence as equality
- Nominal differences
  - $A ::= X \ Y \ Z;$  
  - $B ::= X \ Y \ Z;$
- Structural differences
  - $A ::= X \ Y \ Z;$  
  - $A ::= X \ Z;$
- Deliberately limited comparator is useful
Hamming-based differ

- Resolves structural differences
- Seeks/counts required substitutions
- Yields good results if the transformation suite is replace

Levenshtein-based differ

- Resolves structural differences
- Seeks/counts required single-symbol edits
- Yields good results if the transformation suite is
  - replace
  - permute
  - inject, project

Convergence-based differ

- ‘Cheats’ on undecidability by involving a human
- Do a stupid comparison
- Report a mismatch
- Let a human encode it as transformation
  - ...in a possibly sophisticated framework
- Repeat until equal/equivalent

Grammar convergence

- Source grammar
- Target grammar
- Source grammar
- transformation
- Grammar transformation
- Bidirectional grammar transformation

XBGF

ΞBGF

Signature-based differ

- Heuristic-based human emulator
- Powerful enough for typical local changes
- Case study with 11 grammars:
  - Rascal ADT, ANTLR spec, Prolog DCG, Ecore EMF, JAXB model, Java object model, Rascal syntax def, Python parser, SDF def, TXL def, XML schema

### 7.3 Grammar in ANF

<table>
<thead>
<tr>
<th>Production rule</th>
<th>Production signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(', FLPr, *(FLFun)) )</td>
<td>( {\langle FFun, *\rangle} )</td>
</tr>
<tr>
<td>( p(', FLFun, seq ([str, *(str), FExpr]) )</td>
<td>( {\langle str, 1*\rangle, \langle FExpr, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr, FExpr_1) )</td>
<td>( {\langle FExpr_1, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr, FExpr_2) )</td>
<td>( {\langle FExpr_2, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr, FExpr_3) )</td>
<td>( {\langle FExpr_3, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr, str) )</td>
<td>( {\langle str, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr, int) )</td>
<td>( {\langle int, 1\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr_1, seq ([FExpr, FOp, FExpr]) )</td>
<td>( {\langle FOp, 1\rangle, \langle FExpr, 11\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr_2, seq ([str, *(FExpr)]) )</td>
<td>( {\langle str, 1\rangle, \langle FExpr, *\rangle} )</td>
</tr>
<tr>
<td>( p(', FExpr_3, seq ([FExpr, FExpr, FExpr]) )</td>
<td>( {\langle FExpr, 111\rangle} )</td>
</tr>
</tbody>
</table>

### 7.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[
\begin{align*}
p(', FLPr, *(FLFun)) & \Rightarrow p(', program, +(function)) \\
p(', FLFun, seq ([str, *(str), FExpr]) & \Rightarrow p(', function, seq ([str, +(str), expression])) \\
p(', FExpr, FExpr_1) & \Rightarrow p(', expression, binary) \\
p(', FExpr, FExpr_2) & \Rightarrow p(', expression, apply) \\
p(', FExpr, FExpr_3) & \Rightarrow p(', expression, conditional) \\
p(', FExpr, str) & \Rightarrow p(', expression, str) \\
p(', FExpr, int) & \Rightarrow p(', expression, int) \\
p(', FExpr_1, seq ([FExpr, FOp, FExpr]) & \Rightarrow p(', binary, seq ([expression, operator, expression])) \\
p(', FExpr_2, seq ([str, *(FExpr)]) & \Rightarrow p(', apply, seq ([str, +(expression)])) \\
p(', FExpr_3, seq ([FExpr, FExpr, FExpr]) & \Rightarrow p(', conditional, seq ([expression, expression, expression]))
\end{align*}
\]
Acceptance-based differ

- Take recognisers of different nonterminals
- If they accept the same language,
  - assume them equivalent
- Easily generalisable for partial matches

B. Fischer, R. Lämmel, V. Zaytsev, *Comparison of Context-free Grammars Based on Parsing Generated Test Data* SLE 2011, LNCS 6940, 2012
Acceptance-based differ

B. Fischer, R. Lämmel, V. Zaytsev, *Comparison of Context-free Grammars Based on Parsing*.
*Generated Test Data* SLE 2011, LNCS 6940. 2012
Conclusion
Based on several years of published research

and several years of hacking

(Rascal, Prolog, Python, Haskell, XSLT, ...)

Made at CWI (Centrum Wiskunde & Informatica)

Also presented as a tutorial at MoDELS 2013

http://grammarware.github.io/lab
include lproject://grammarlab/zoo/csharp/ecma-334-1.gluel.

DeYaccifyAll.
UnchainAll.
InlinePlus.

inline using-alias-directive.
inline using-namespace-directive.

factor ("using" identifier "=" namespace-or-type-name ";" | "using" namespace-name ";")
to ("using" (namespace-name | identifier "=" namespace-or-type-name) ";")
in using-directive.

extract using-directive-insides ::= namespace-name | (identifier "=" namespace-or-type-name);
globally.

inline using-directive.

splitT "," into "," "," in global-attribute-section.

factor
  ( "[" global-attribute-target-specifier attribute-list "]"
  | "[" global-attribute-target-specifier attribute-list "," "," "]")
to ("[" global-attribute-target-specifier (attribute-list | attribute-list ",") "]")
in global-attribute-section.

inline global-attribute-target-specifier.
inline global-attribute-target.

extract global-attribute-section-insides ::= attribute-list | attribute-list ","; globally.

inline class-declaration.
inline struct-declaration.
inline interface-declaration.
inline enum-declaration.
inline delegate-declaration.

rename class-modifier to modifier globally.

unite struct-modifier with modifier.
Evolution is a thing

Imperative is easy and weak

Declarative is complex and powerful

Ideally, we want easy + support

Various approaches


Questions?