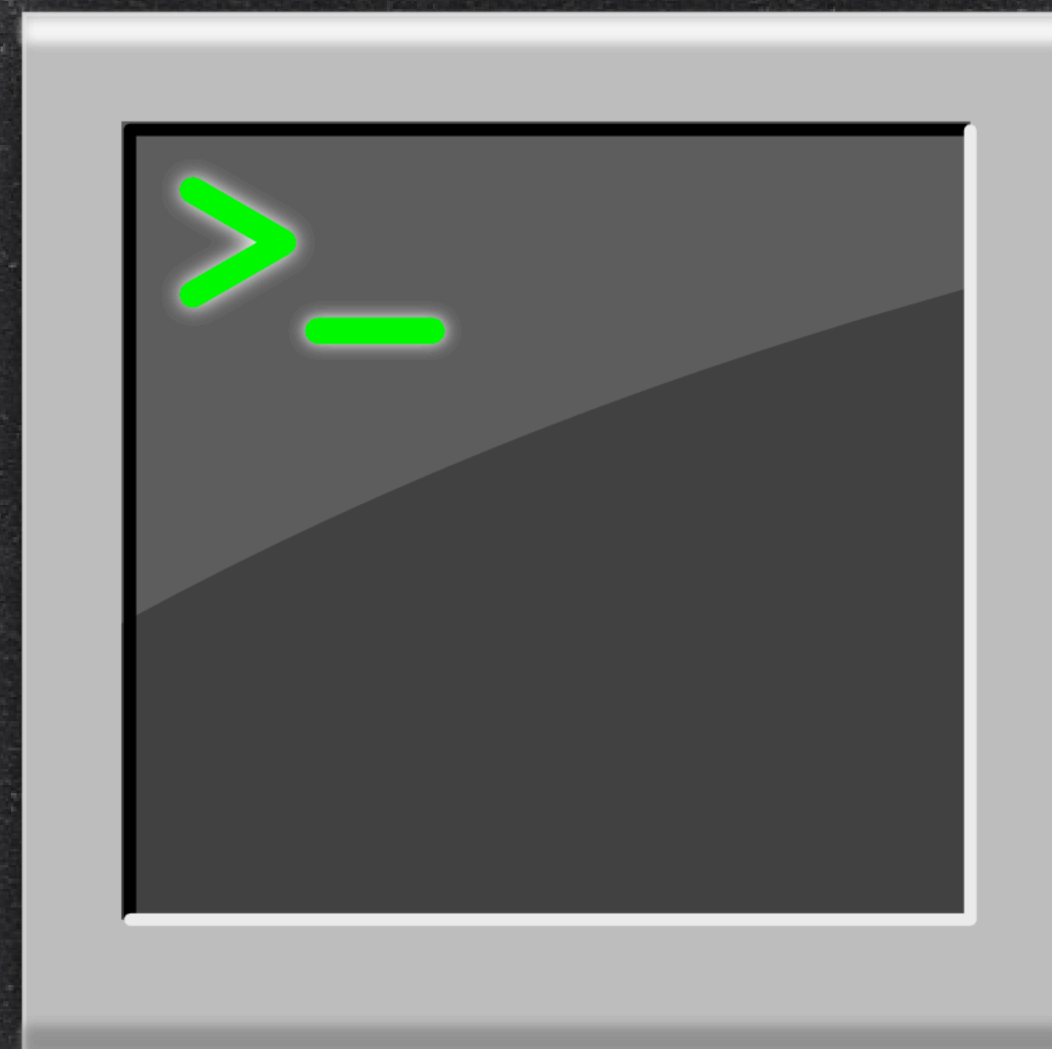
Imperative and Declarative Views on Software Language Evolution

Vadim Zaytsev
Universiteit van Amsterdam
NLFP 2014

Introduction

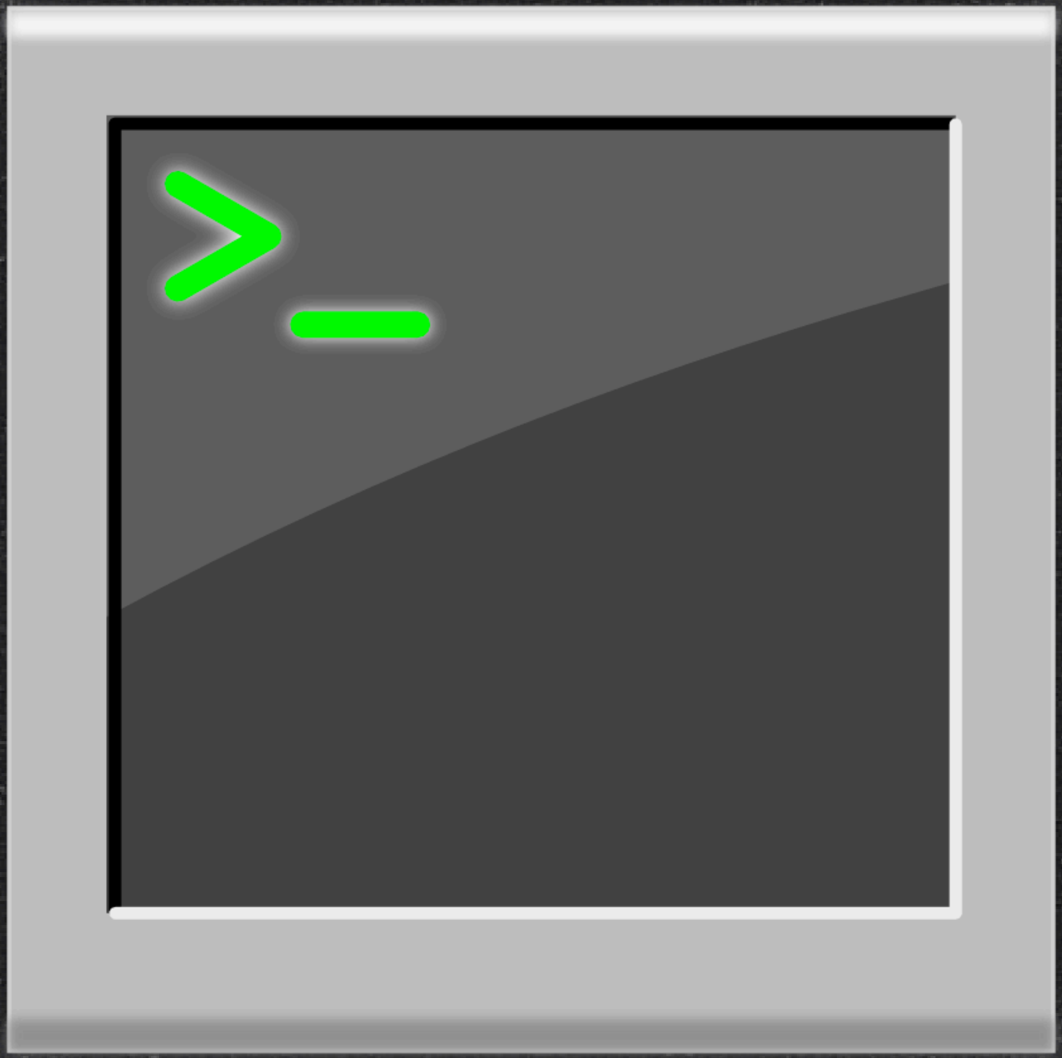
- Universiteit van Amsterdam (2013–2014)
- Centrum Wiskunde & Informatica (2010–2013)
- Universität Koblenz-Landau (2008–2010)
- Vrije Universiteit Amsterdam (2004–2008)
- Universiteit Twente (2002–2004)
- Rostov State University (1998–2003)



Vadim Zaytsev

Introduction

- Haskell (2013–2014)
- Rascal (2010–2013)
- Prolog (2008–2010)
- Smalltalk (2004–2008)
- XSLT (2002–2004)
- Python (1998–2003)

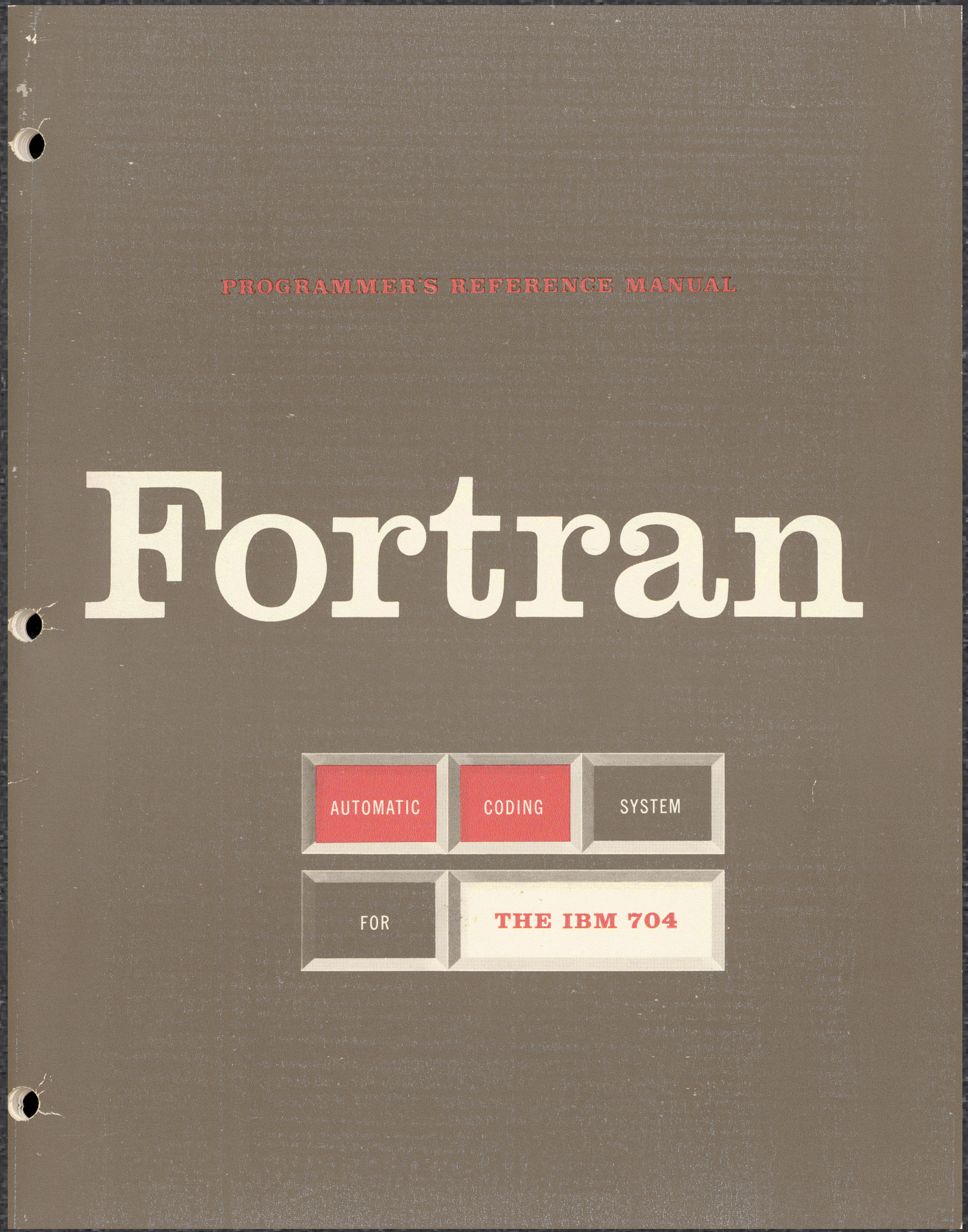


Vadim Zaytsev

Part I
SLE background

Software Languages

- Programming languages



Software Languages

- Programming languages
- Functional languages

The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications,* page 681.

"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 necessarily 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 (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for 'profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications 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 computations called Fortran. This same group designed the first 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 1966.

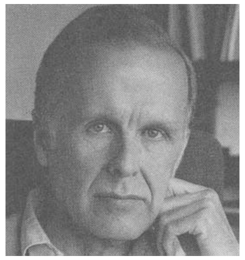
During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 60. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form" to recognize the further contributions by Peter Naur 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 linguistics. 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 enormous, 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 structured data, are often nonrepetitive 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.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

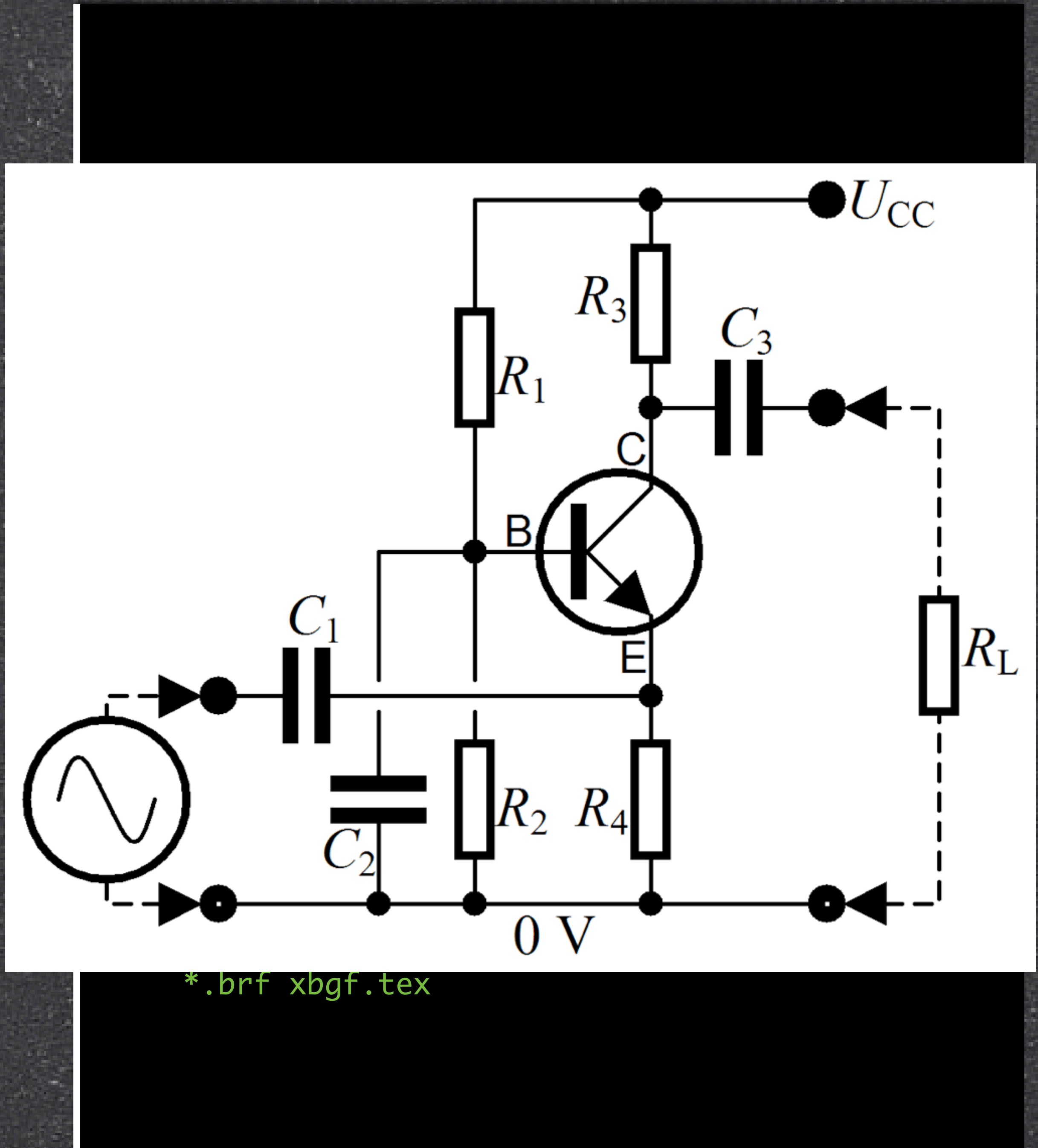
Software Languages

- Programming languages
- Functional languages
- Declarative languages

```
build:
	pdflatex paper
	bibtex paper
	pdflatex -interaction=batchmode paper
	pdflatex -interaction=batchmode paper
	open paper.pdf

rebuild:
	make clean
	make chapter1
	make chapter3
	make chapter4
	make chapter5
	make chapter6
	make build

clean:
	rm -f *~ *.aux *.bbl *.blg *.lo? *.toc
*.brf xbgf.tex
```

Software Languages

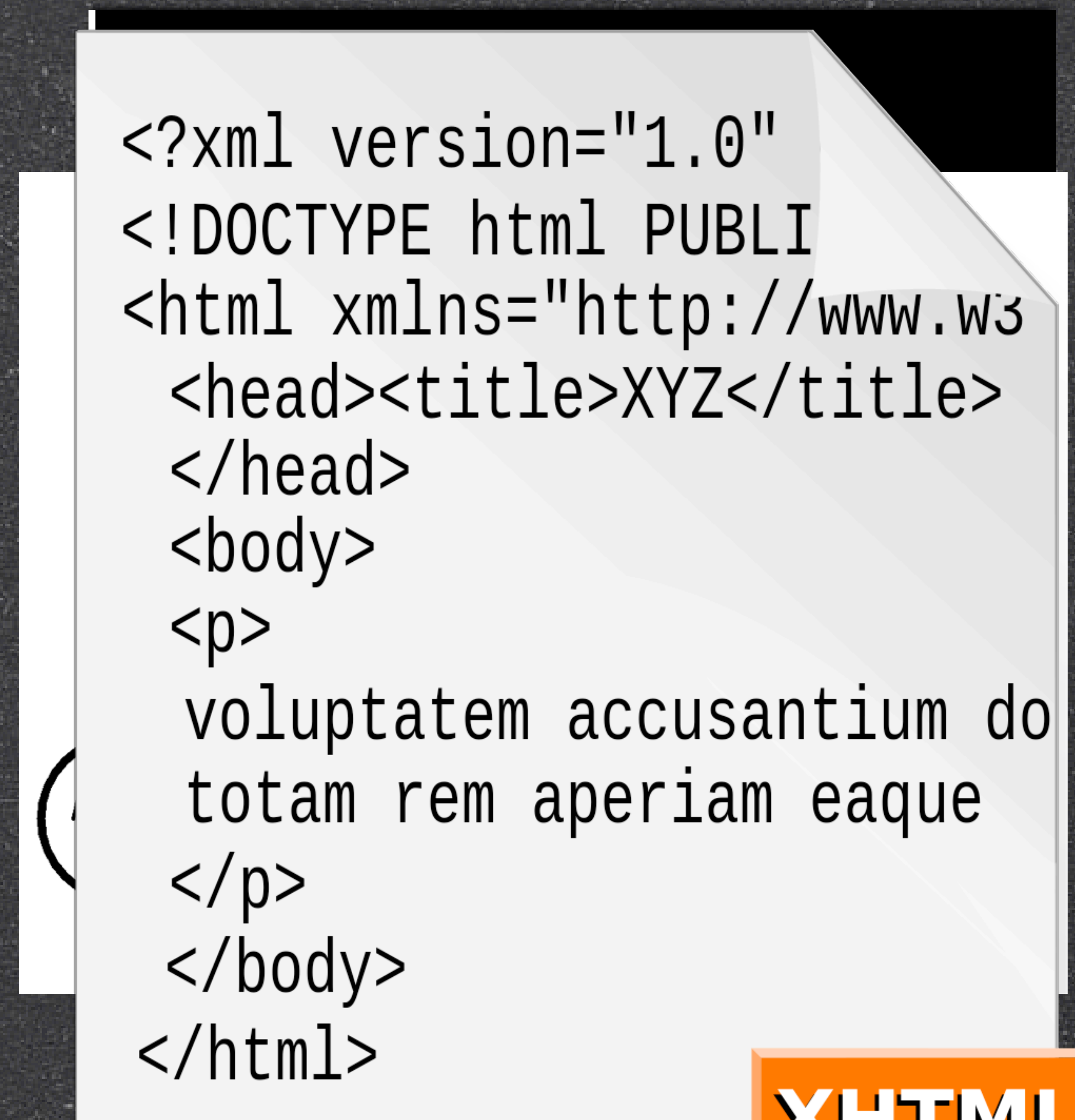
- Programming languages
- Functional languages
- Declarative languages
- Modelling languages



http://en.wikipedia.org/wiki/File:Common_Base_amplifier.png

Software Languages

- Programming languages
- Functional languages
- Declarative languages
- Modelling languages
- Markup languages
- …



http://commons.wikimedia.org/wiki/File:XHTML.svg

http://en.wikipedia.org/wiki/File:Common_Base_amplifier.png

Software Language Evolution

- Language → next version
  - more features
  - backward compatibility
- DSL → 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 undecideable

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)
```

Grammar example (ADT)

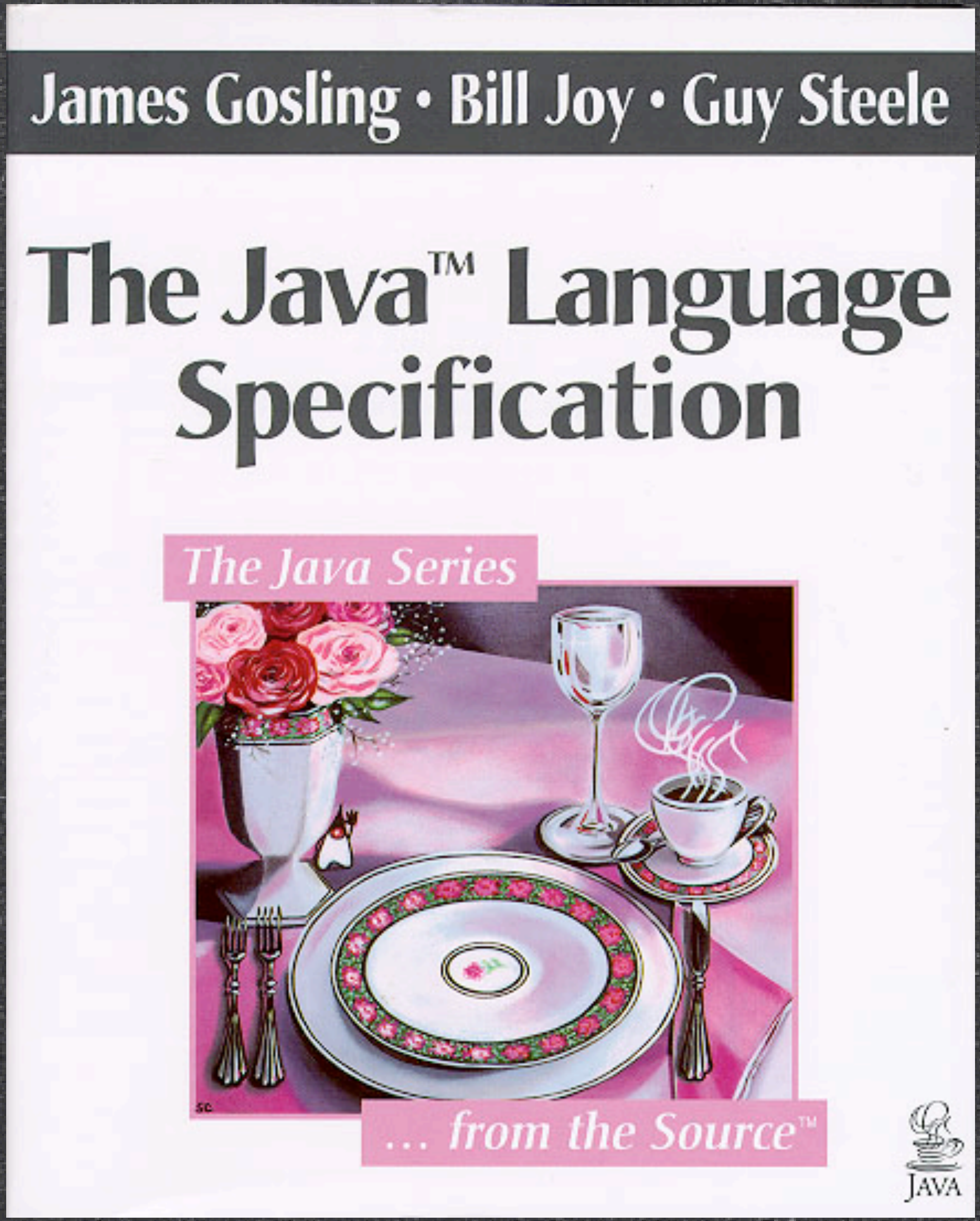
Types.hs — haskell

```
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;
```

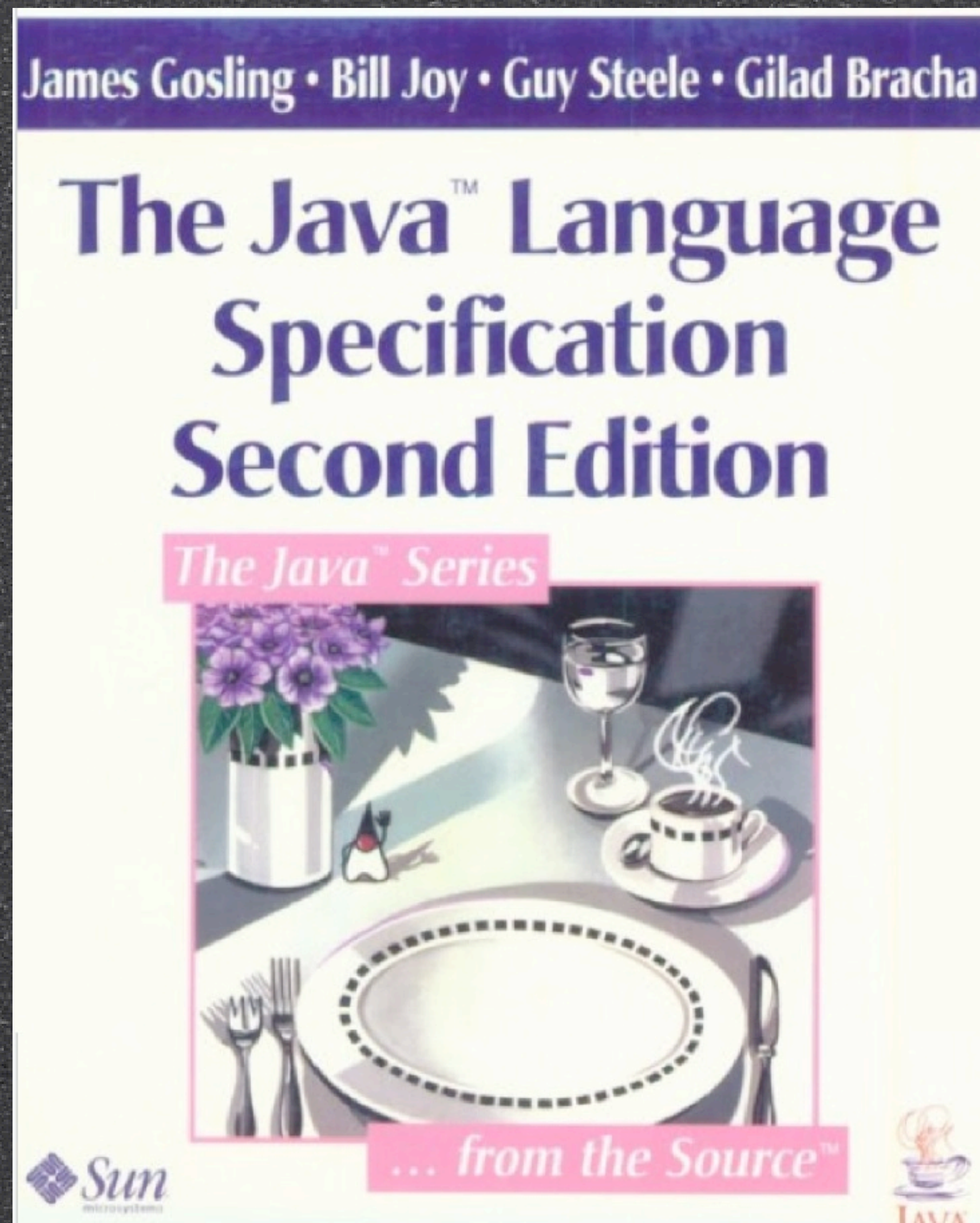
Part II
Imperative View

Imperative view
on software language evolution

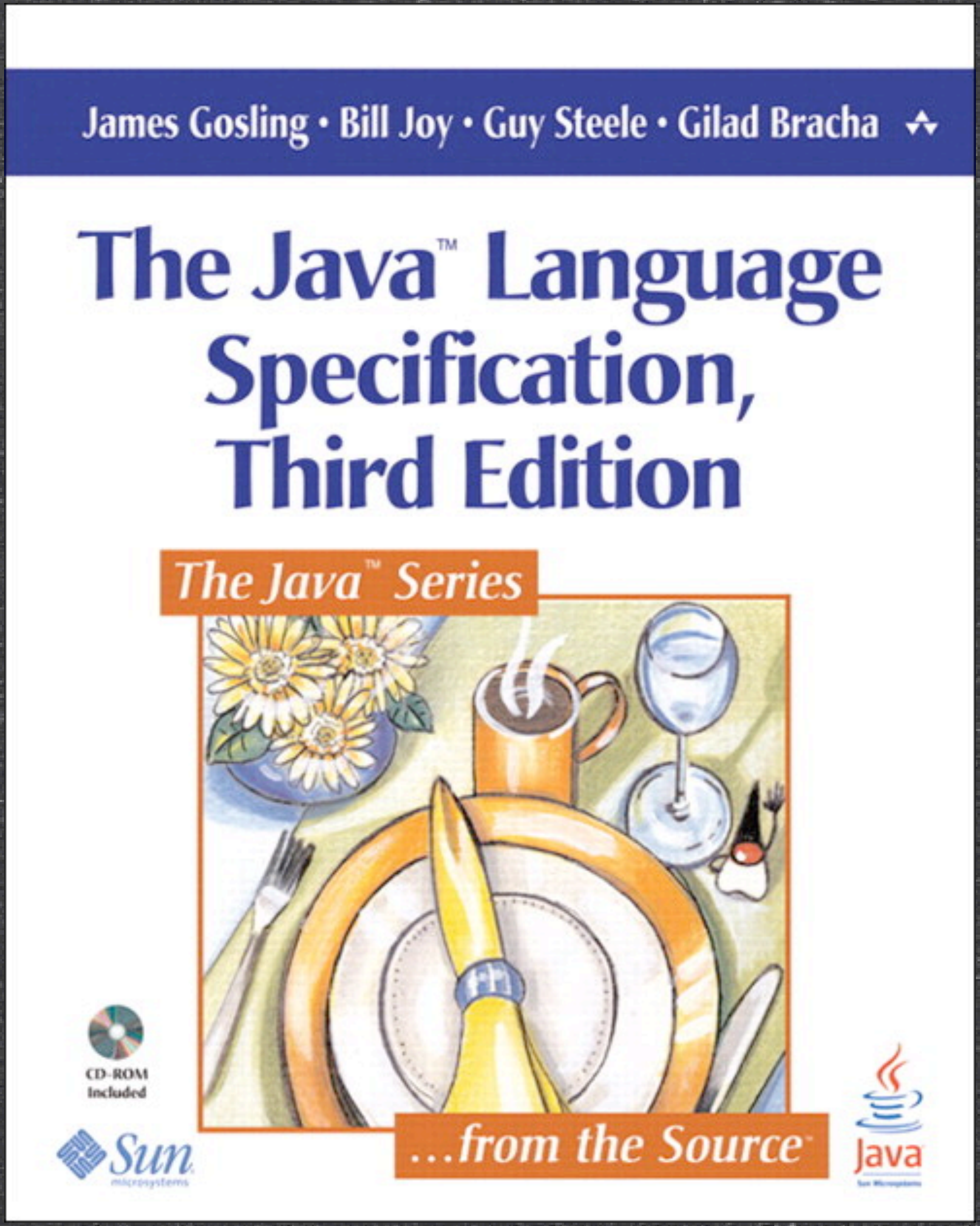
Imperative example



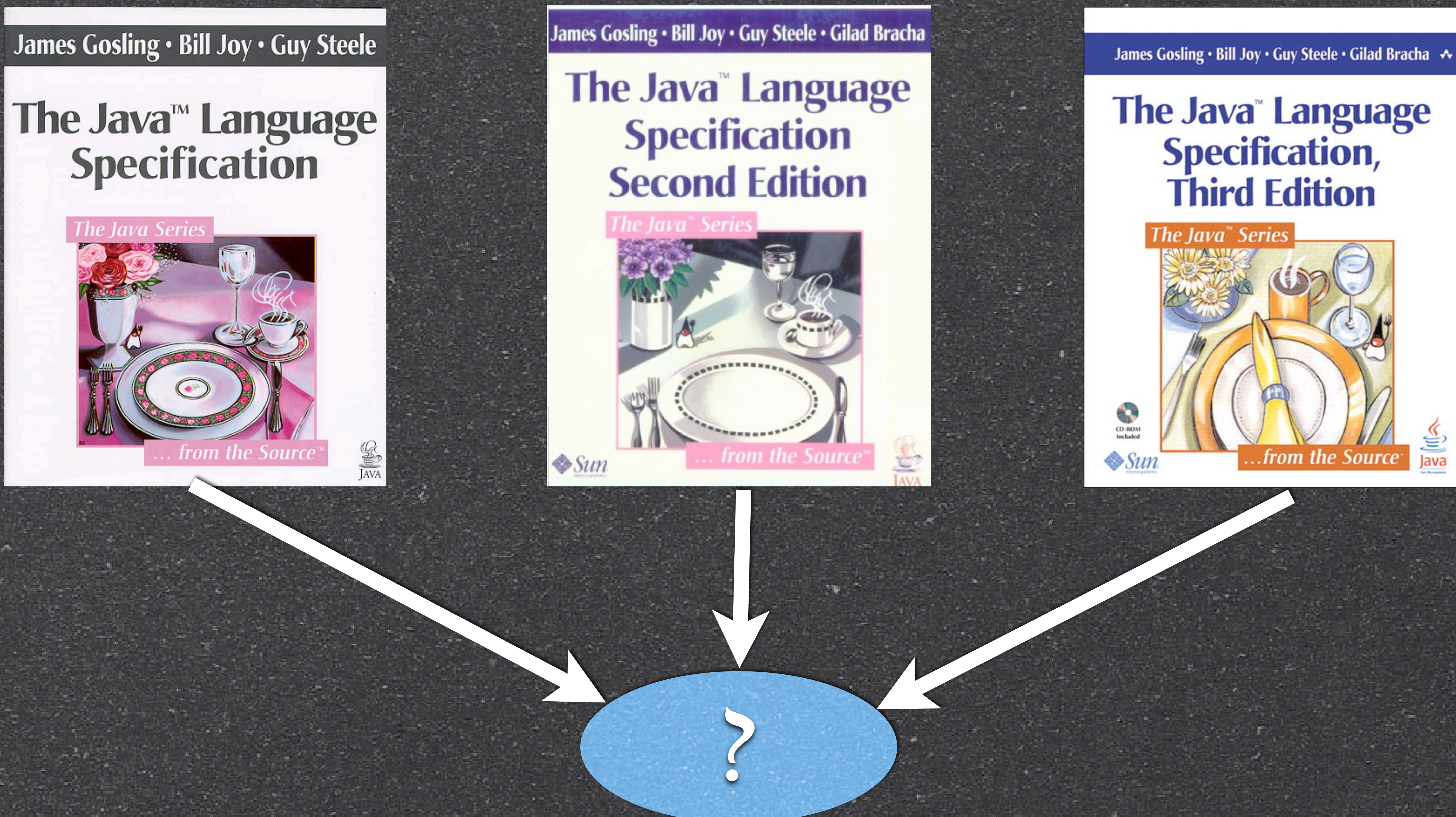
Imperative example



Imperative example



Imperative example



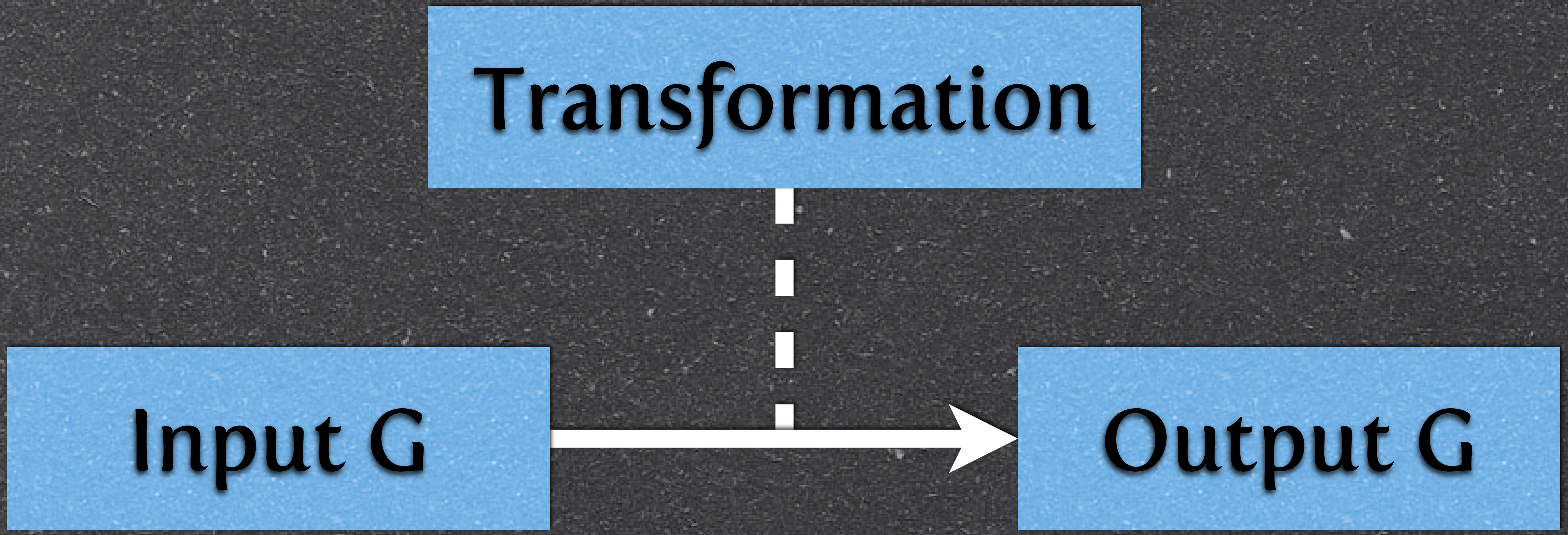
R. Lämmel, V. Zaytsev, *Recovering Grammar Relationships for the Java Language Specification*. SQJ, 2011.

Grammar differences

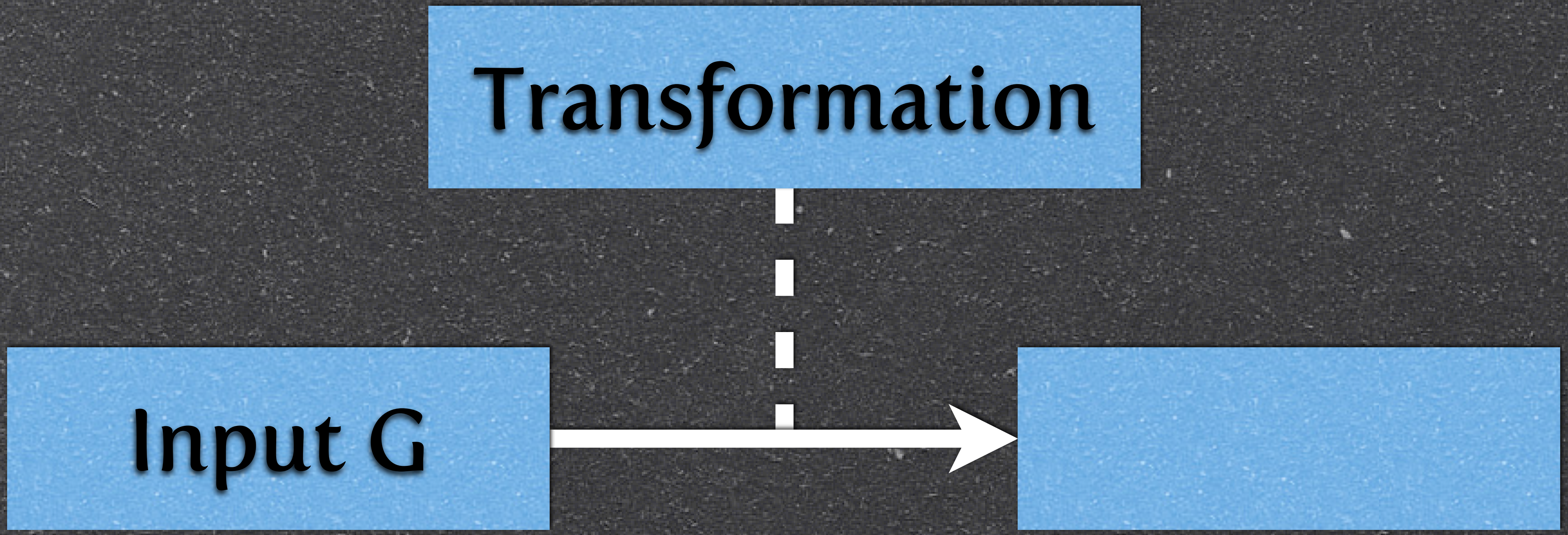
- intended vs. accidental
- result of grammar adaptation
- result of grammar evolution
- idiosyncrasies thanks to metanotation
- idiosyncrasies thanks to parsing technology
- presentation and understandability
- misspelling
- …etc

Part III
Declarative View

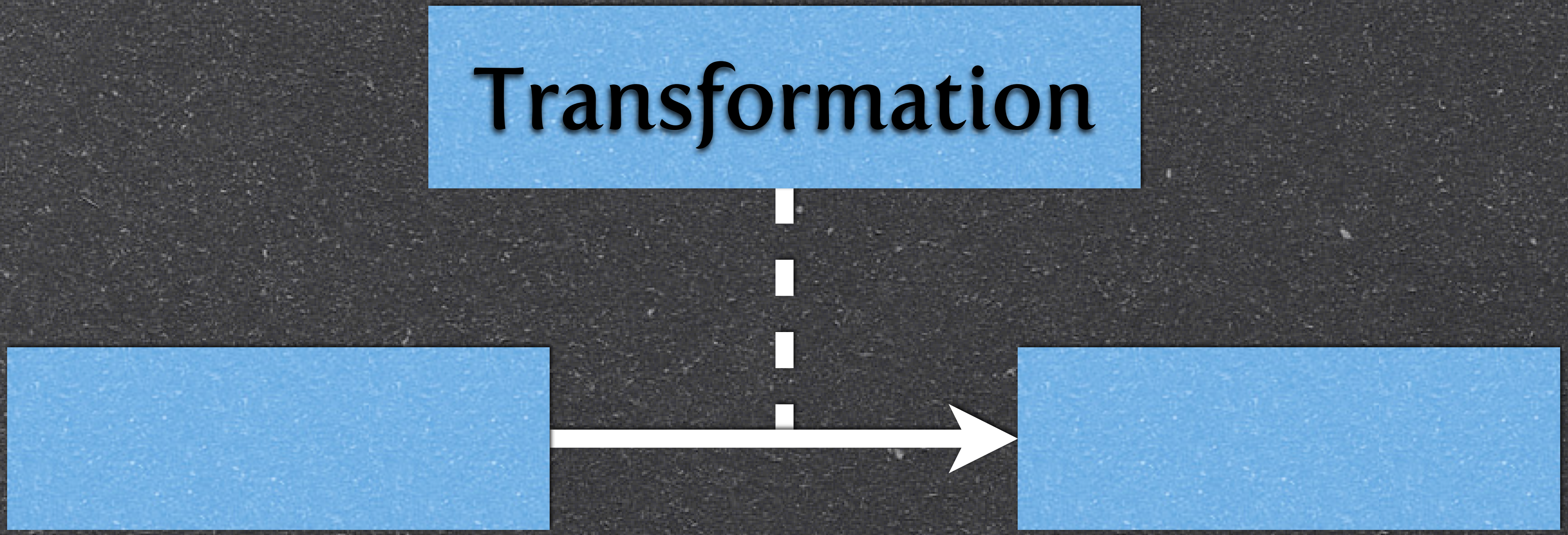
Declarative view
on software language evolution



Declarative view
on software language evolution



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;
expr : expr;

abridge

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

R. Lämmel, V. Zaytsev, An Introduction to Grammar Convergence. IFM 2009, LNCS 5423.

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

Grammar mutations

- distribute ⊢ DistributeAll
- eliminate ⊢ EliminateTop
- concatT ⊢ ConcatAllT
- inline ⊢ InlineLazy
- renameN ⊢ RenameNUpperDash2CamelNone
- define ⊢ DefineAll([p_i])

V. Zaytsev, Software Language Engineering by Intentional Rewriting. SQM 2014.

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

Bridging/mapping

- Both approaches have (dis)advantages
- Declarative → imperative
  - easy, if the input is given
- Imperative → 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`

R. W. Hamming, "Error Detecting And Error Correcting Codes",
Bell System Technical Journal 29 (2): 147–160, MR 0035935. 1950.

Levenshtein-based differ

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

V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," Soviet Physics Doklady, vol. 10, no. 8, pp. 707–710, 1966.

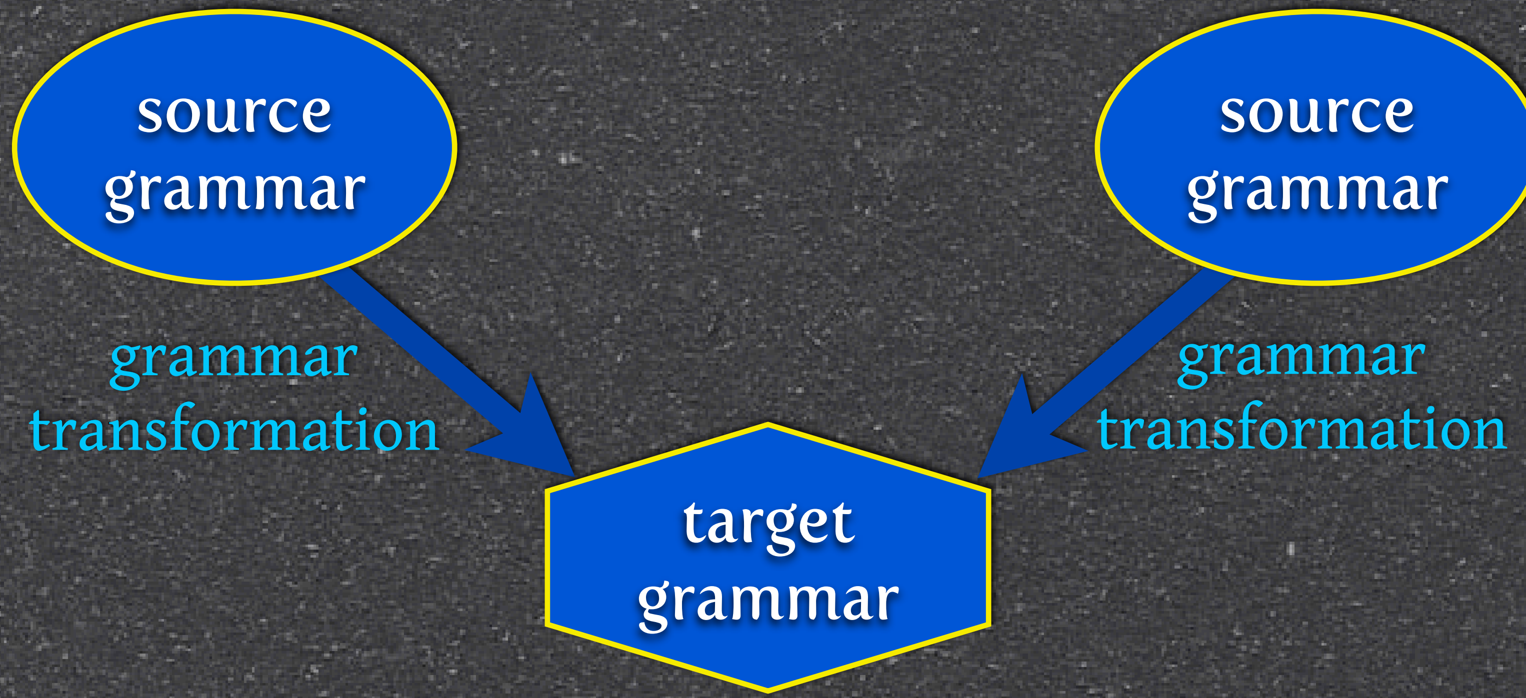
Convergence-based differ

- ‘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

R. Lämmel, V. Zaytsev, *An Introduction to Grammar Convergence*. IFM 2009, LNCS 5423.

Grammar convergence

XBGF



ΞBGF

= relationship

V. Zaytsev, Language Evolution, Metasyntactically. EC-EASST 49, 2012.

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

V. Zaytsev, *Guided Grammar Convergence*. SLE Poster, CEUR, 2013.

7.3 Grammar in ANF

Production rule	Production signature
$\mathrm{p}\,(`', \mathit{FLPrg}, *(\mathit{FLFun}))$	$\{\langle \mathit{FLFun}, * \rangle\}$
$\mathrm{p}\,(`', \mathit{FLFun}, \mathrm{seq}\,([\mathit{str}, *(\mathit{str})\,, \mathit{FLExpr}]))$	$\{\langle \mathit{str}, 1* \rangle, \langle \mathit{FLExpr}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_1})$	$\{\langle \mathit{FLExpr_1}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_2})$	$\{\langle \mathit{FLExpr_2}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_3})$	$\{\langle \mathit{FLExpr_3}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{str})$	$\{\langle \mathit{str}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{int})$	$\{\langle \mathit{int}, 1 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr_1}, \mathrm{seq}\,([\mathit{FLExpr}, \mathit{FLOp}, \mathit{FLExpr}]))$	$\{\langle \mathit{FLOp}, 1 \rangle, \langle \mathit{FLExpr}, 11 \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr_2}, \mathrm{seq}\,([\mathit{str}, *(\mathit{FLExpr})]))$	$\{\langle \mathit{str}, 1 \rangle, \langle \mathit{FLExpr}, * \rangle\}$
$\mathrm{p}\,(`', \mathit{FLExpr_3}, \mathrm{seq}\,([\mathit{FLExpr}, \mathit{FLExpr}, \mathit{FLExpr}]))$	$\{\langle \mathit{FLExpr}, 111 \rangle\}$

7.4 Nominal resolution

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

$$
\begin{aligned}
\mathrm{p}\,(`', \mathit{FLPrg}, *(\mathit{FLFun})) &\quad\Bumpeq\quad \mathrm{p}\,(`', \mathit{program}, +(\mathit{function}))\\
\mathrm{p}\,(`', \mathit{FLFun}, \mathrm{seq}\,([\mathit{str}, *(\mathit{str})\,, \mathit{FLExpr}])) &\quad\Bumpeq\quad \mathrm{p}\,(`', \mathit{function}, \mathrm{seq}\,([\mathit{str}, +(\mathit{str})\,, \mathit{expression}]))\\
\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_1}) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{expression}, \mathit{binary})\\
\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_2}) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{expression}, \mathit{apply})\\
\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{FLExpr_3}) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{expression}, \mathit{conditional})\\
\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{str}) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{expression}, \mathit{str})\\
\mathrm{p}\,(`', \mathit{FLExpr}, \mathit{int}) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{expression}, \mathit{int})\\
\mathrm{p}\,(`', \mathit{FLExpr_1}, \mathrm{seq}\,([\mathit{FLExpr}, \mathit{FLOp}, \mathit{FLExpr}])) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{binary}, \mathrm{seq}\,([\mathit{expression}, \mathit{operator}, \mathit{expression}]))\\
\mathrm{p}\,(`', \mathit{FLExpr_2}, \mathrm{seq}\,([\mathit{str}, *(\mathit{FLExpr})])) &\quad\Bumpeq\quad \mathrm{p}\,(`', \mathit{apply}, \mathrm{seq}\,([\mathit{str}, +(\mathit{expression})]))\\
\mathrm{p}\,(`', \mathit{FLExpr_3}, \mathrm{seq}\,([\mathit{FLExpr}, \mathit{FLExpr}, \mathit{FLExpr}])) &\quad\bumpeq\quad \mathrm{p}\,(`', \mathit{conditional}, \mathrm{seq}\,([\mathit{expression}, \mathit{expression}, \mathit{expression}]))
\end{aligned}
$$

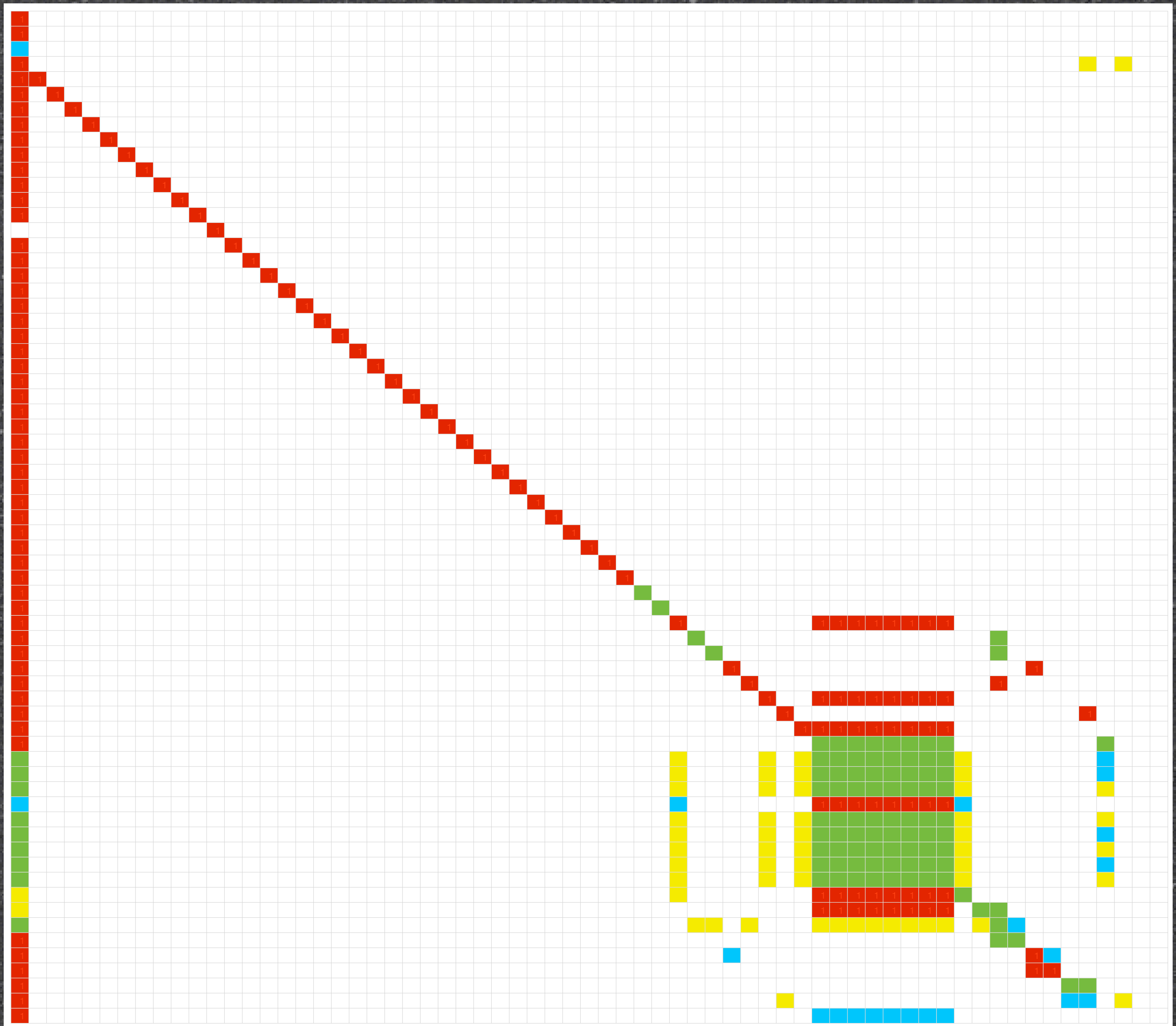
V. Zaytsev, Guided Grammar Convergence. arXiv:1207.6541v1 [cs.PL]. 2012.

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

GRAMMARLAB

- 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 |project://grammarlab/zoo/csharp/ecma-334-1.glue|.
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.
```

Imperative vs Declarative

- Evolution is a thing
- Imperative is easy and weak
- Declarative is complex and powerful
- Ideally, we want easy + support
  - various approaches
- Vadim Zaytsev, http://grammarware.net
- Questions?