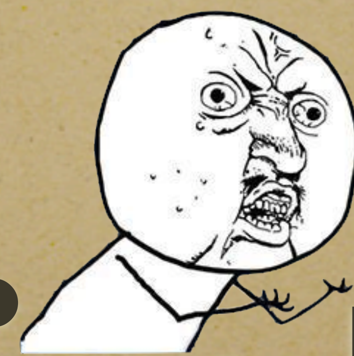
Bidirectional Grammar Transformations

Software Languages Team, Universität Koblenz-Landau
Vadim Zaytsev, SWAT, CWI

In a nutshell

- Grammars in the broad sense
  - protocols, metamodels, schemata, ontologies, structures
- Grammars evolve
  - language evolution, error fixes, dialects, etc
- Systematic way
  - programmable transformation steps
  - operator suite
  - bidirectionality



Motivation

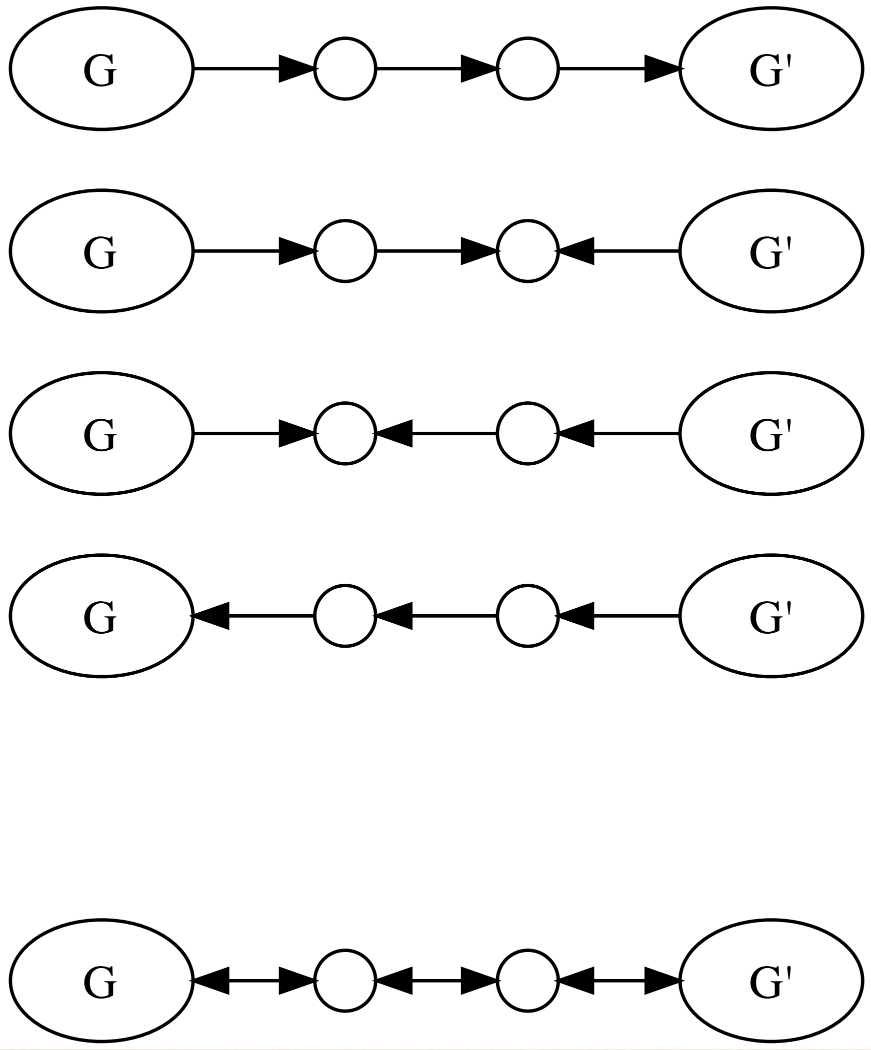
Why transformation?

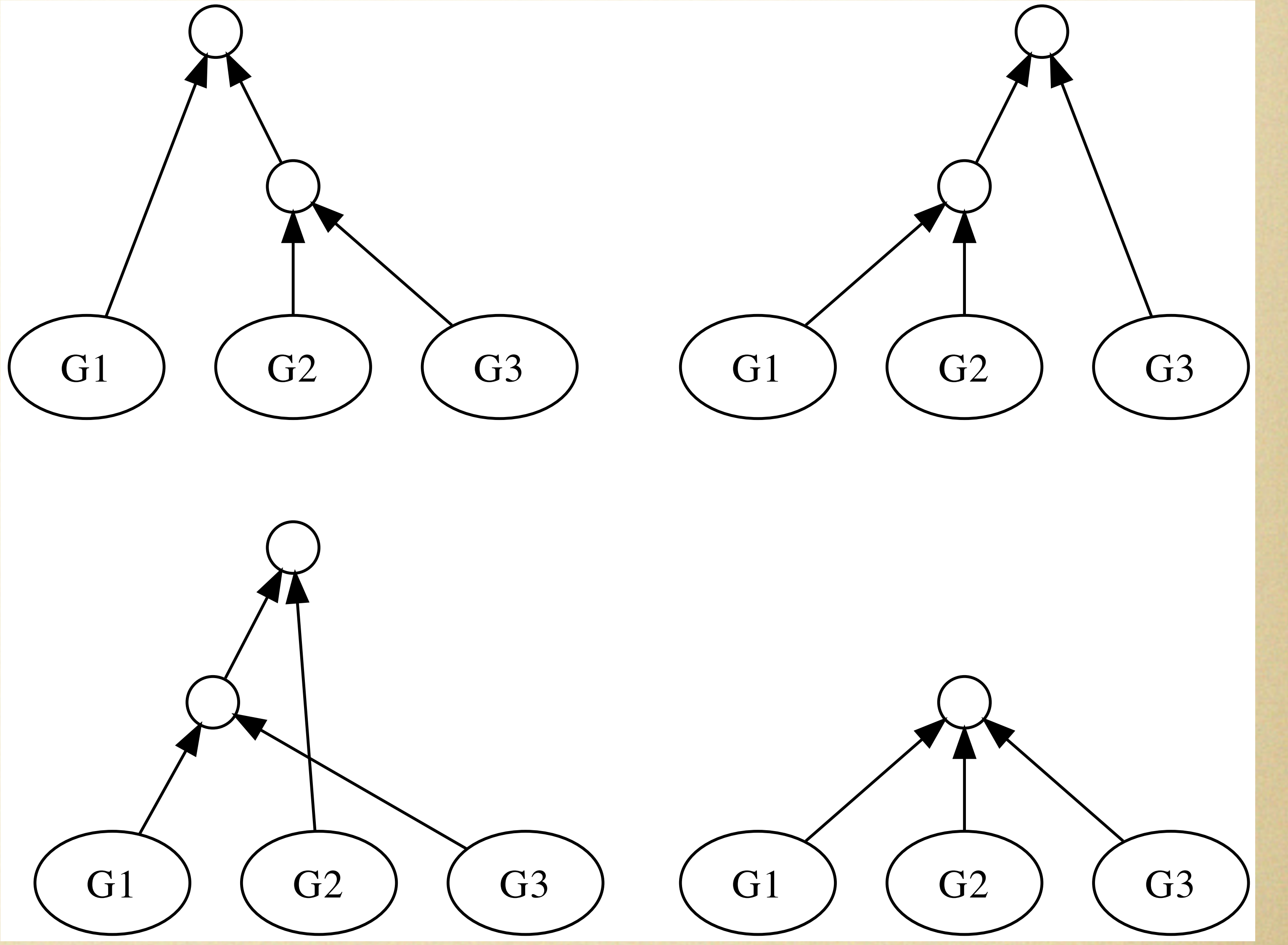
- Edit as text
  - no (technical) problems during editing
  - some problems afterwards
  - no automation
  - no replication
- Program transformations
  - somewhat more complex technically
  - more maintainable, reproducible, transformable, …

Why bidirectional?

- Sweet spot between
  - functional approach
    - $y = f(x)$
  - declarative approach
    - $f(x,y) :- \ldots$
- Fixed application order, direction is flexible

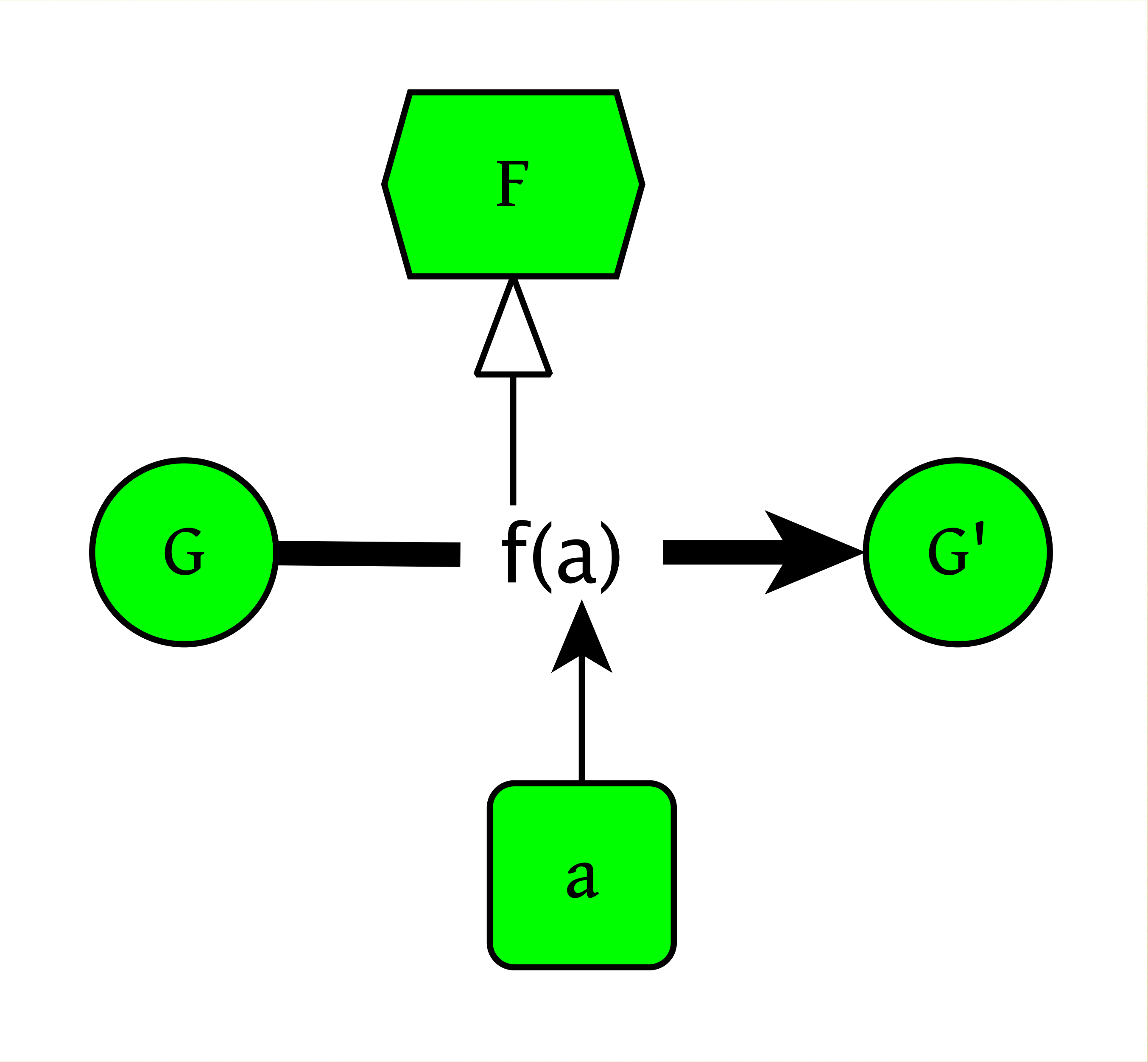
Bidirectionality profits



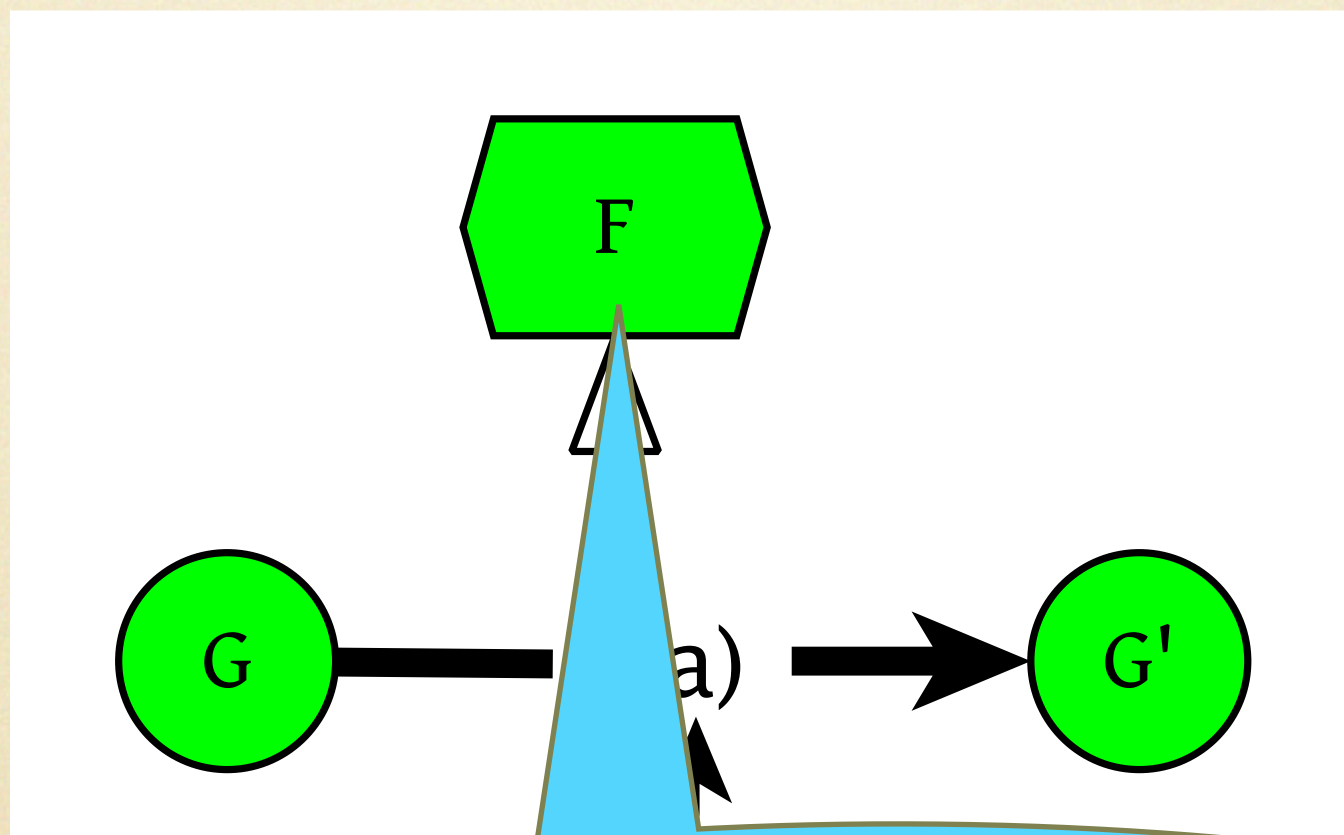


Unidirectional Transformations

Transformation components



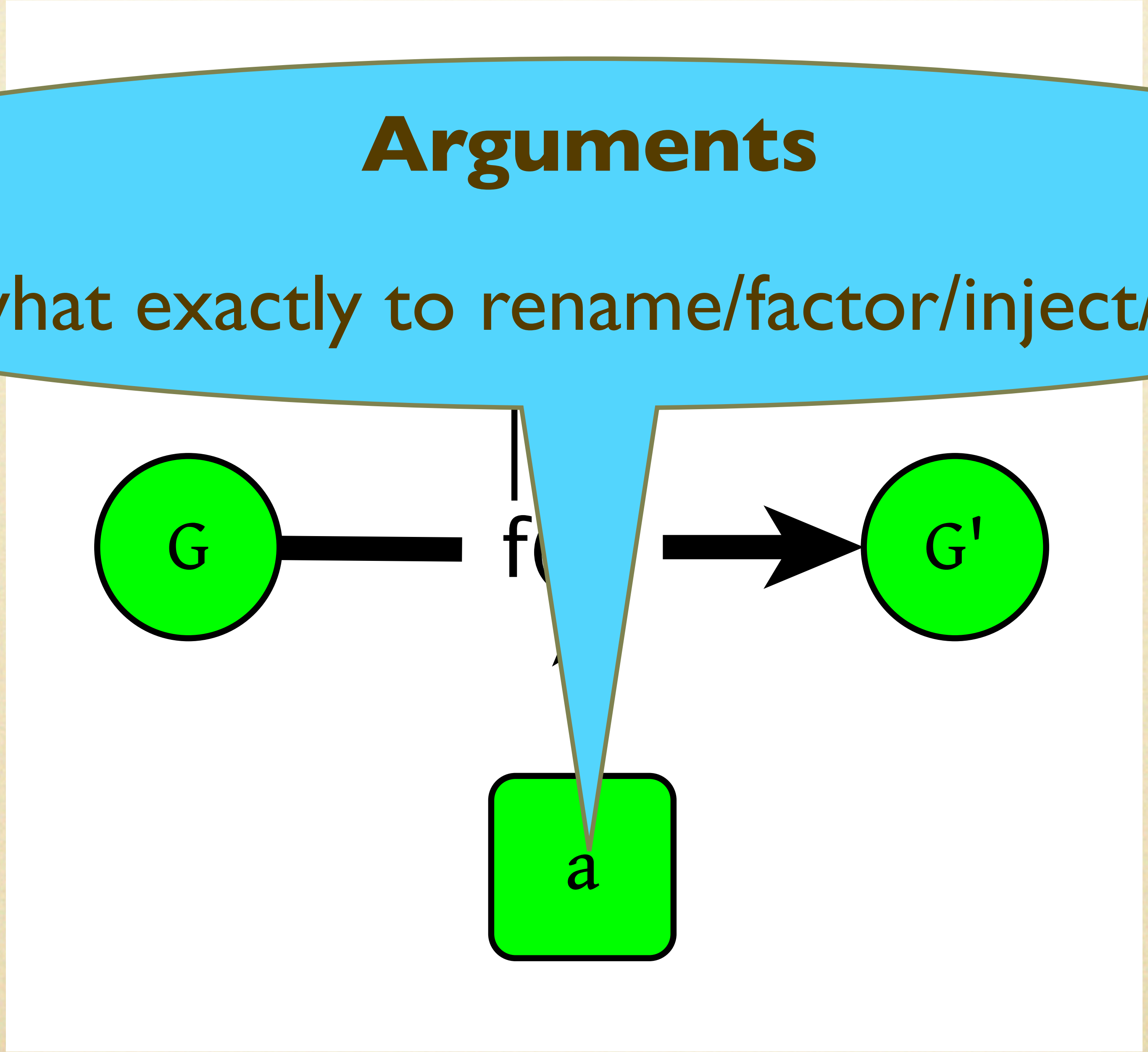
Transformation components



Operator

- known semantics, well-defined algorithm
  - rename, fold, factor, inject, remove, …

Transformation components



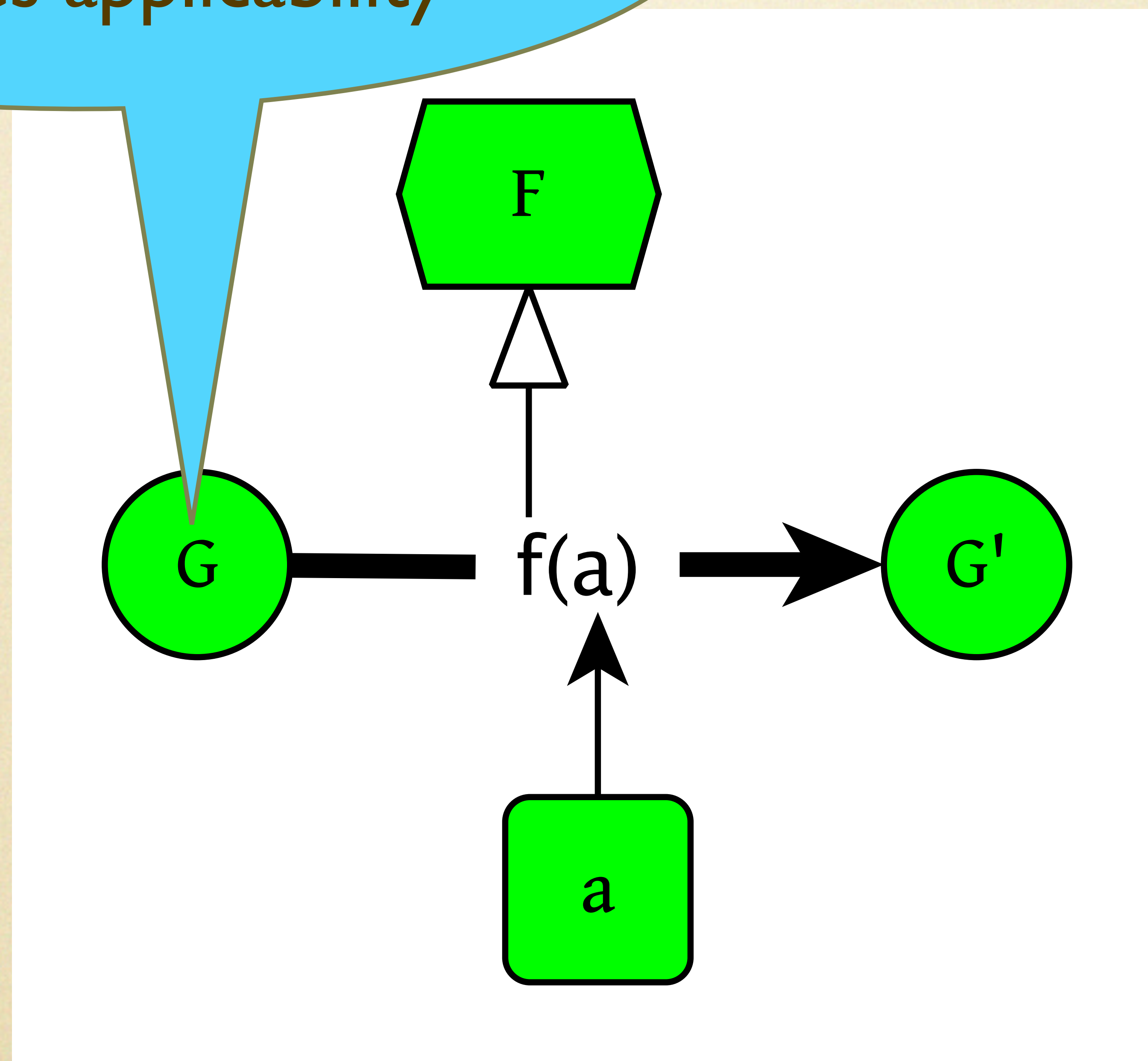
Arguments

- what exactly to rename/factor/inject/…?

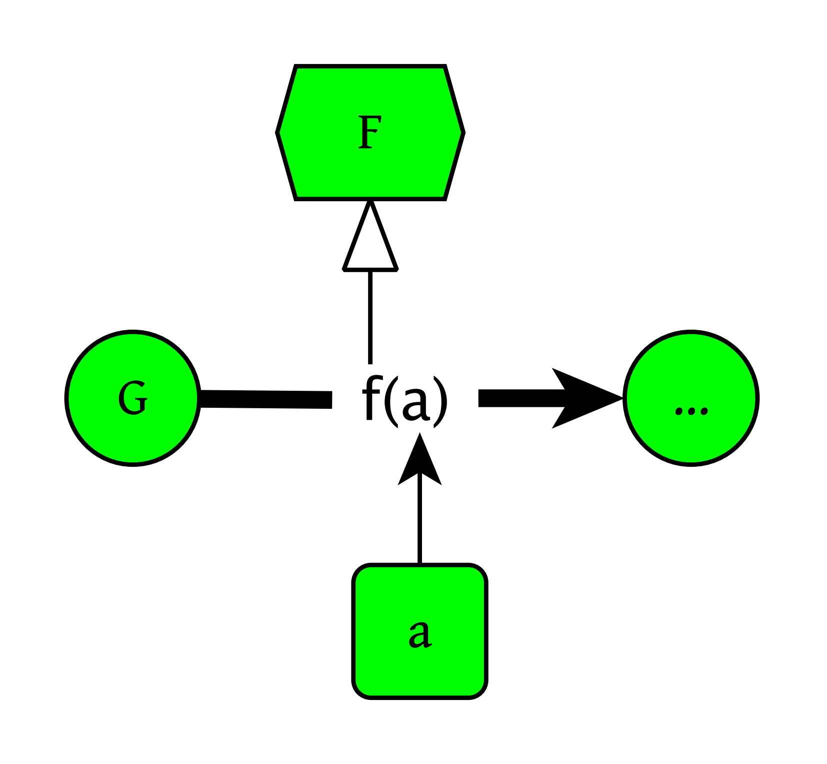
T n components

Input grammar

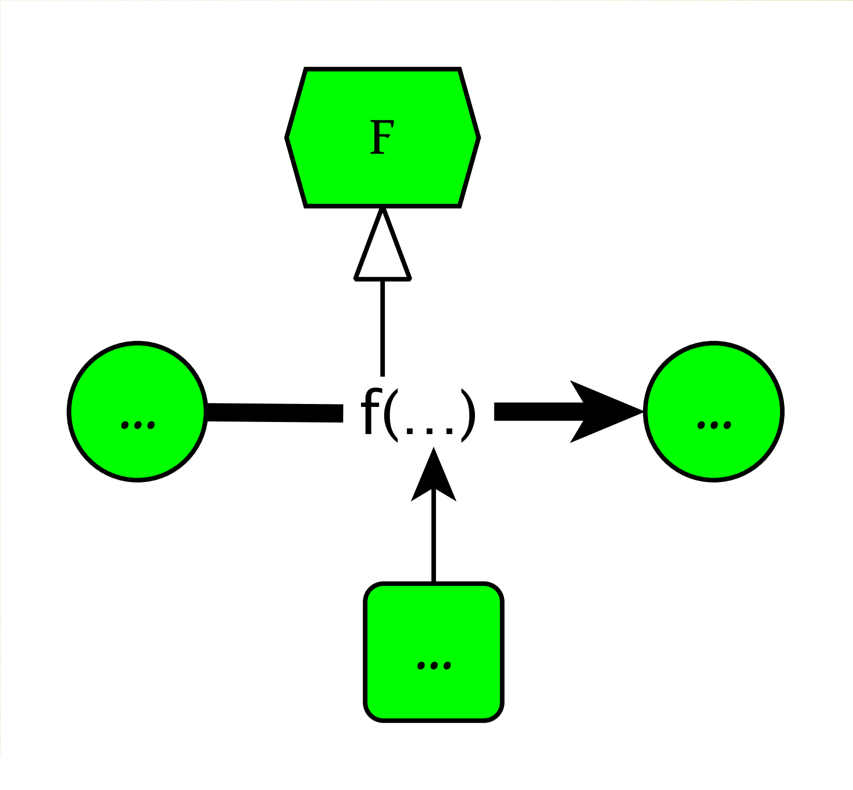
- determines applicability

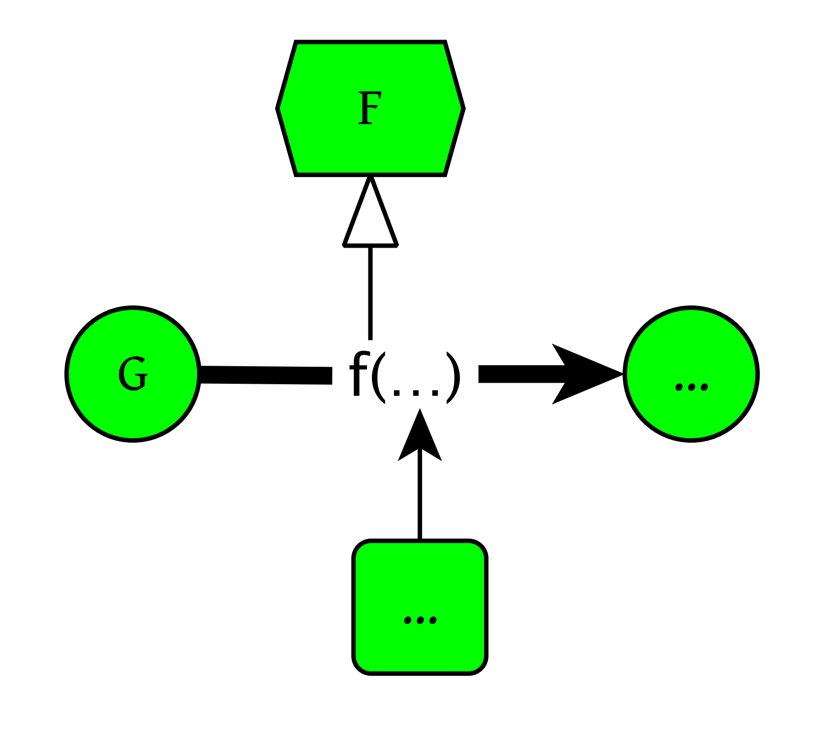


Variations on components



other variations possible





V. Zaytsev, *Maintenance and Evolution of Grammarware by Grammar Transformation*. IPA SD 2012.

CWI

Example

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

A fragment of concrete syntax.
What if we want to derive the abstract syntax?

CWI

Example

Need to project away ‘(’ & ‘)’

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

Need to merge “expr” & “atom”

Alternative needs to go entirely

A fragment of concrete syntax.
What if we want to derive the abstract syntax?

A transformation sequence

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;

XBGF Operator Suite

$$L(G_1) = L(G_2)$$

- Semantics–preserving (refactoring)
  - rename, import, introduce, eliminate
  - fold, unfold, extract, inline
  - factor, distribute, horizontal, <u>vertical</u>
  - yaccify, deyaccify, massage
  - designate, unlabel
  - <u>abridge</u>, detour
  - ...

CWI

XBGF Operator Suite

- Semantics-in/decreasing
  - appear, disappear
  - narrow, widen
  - add, remove
  - upgrade, downgrade
  - <u>unite</u>
  - ...

$$L(G_1) \subseteq L(G_2)$$

$$\vee$$

$$L(G_2) \subseteq L(G_1)$$

XBGF Operator Suite

$$L(G_1) \not\subseteq L(G_2)$$

$$\wedge$$

$$L(G_2) \not\subseteq L(G_1)$$

- Semantics-revising
  - undefine, define
  - inject, project, permute
  - <u>abstractize</u>, concretize
  - replace, redefine

Bidirectional Transformations

CWI

ΞBGF Operator Suite

- Pairs already formed
  - chain — unchain
  - add — remove
  - fold — unfold
  - …
- Implicit pairs
  - factor — factor
  - massage — massage
  - replace — replace
  - …

CWI

ΞBGF Operator Suite

- Incomplete pairs
  - rassoc — ???
  - unite — ???
  - equate — ???
  - …
- Asymmetrical pairs
  - designate(prod) — unlabel(label)
  - deyaccify(nt) — yaccify(prods)
  - eliminate(nt) — introduce(prods)
  - …

ΞBGF Operator Suite

- Remaining problems
  - Lack of symmetry in execution
    - unfold(A:ε) — fold???
    - unfold(A:b in bAb) — fold???
  - …
- Hence, bidirectionality and not bijection

To summarise

- Grammar transformation steps describe software language evolution
- Programmable grammar transformation operators
- XBGF: unidirectional
- ΞBGF: bidirectional
- SLPS: implementations in Prolog and Rascal
- http://grammarware.github.com

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg

Questions?

vadim@grammarware.net