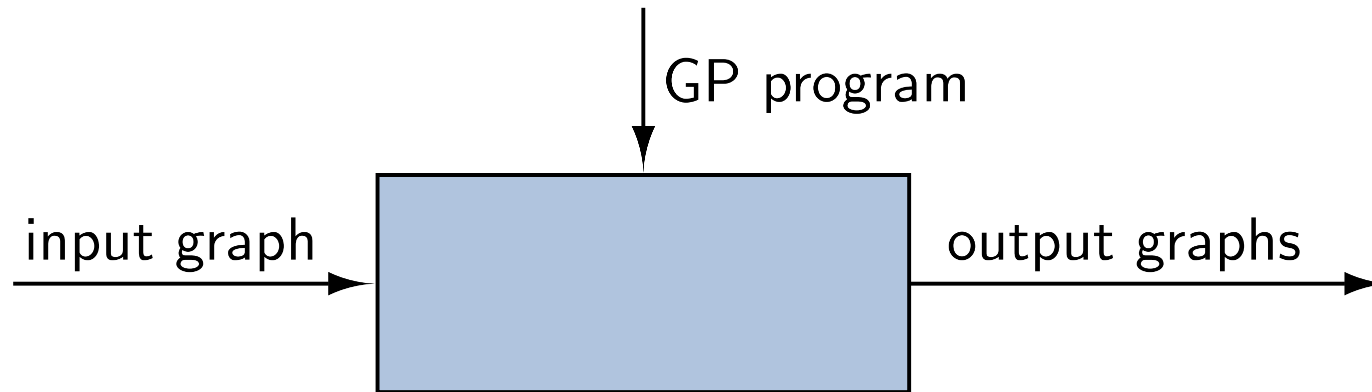
The Graph Programming Language GP

Detlef Plump

The University of York

GP (Graph Programs)



GP (Graph Programs)

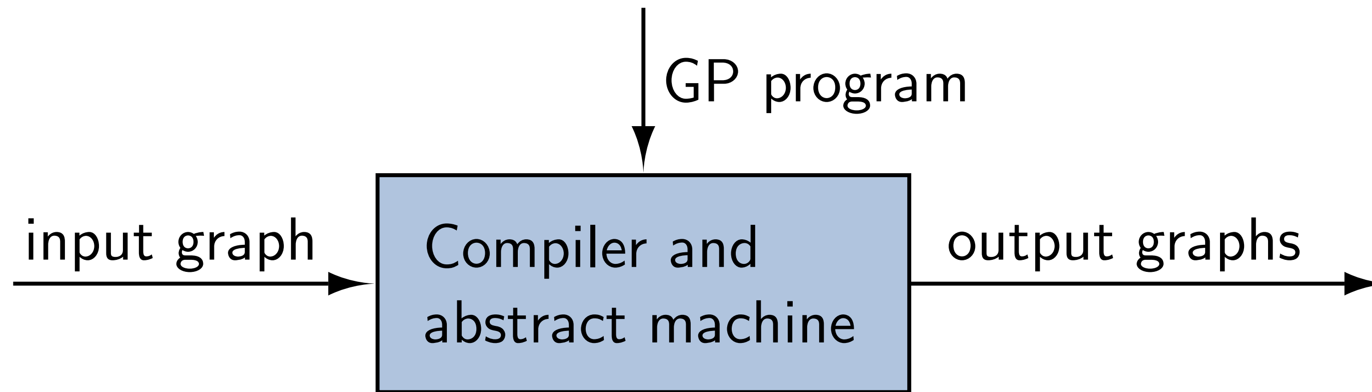
GP program

input graph

output graphs

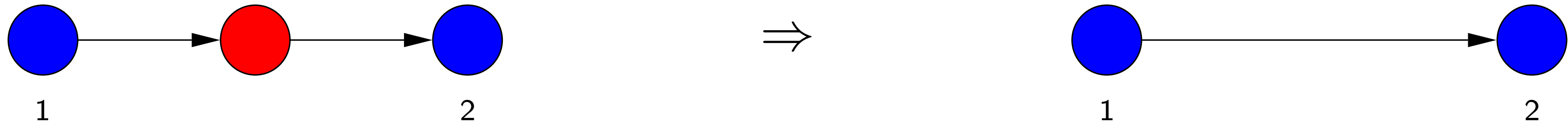
- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Computationally complete
- Formal semantics

GP (Graph Programs)

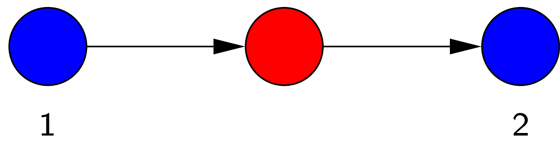


- Based on graph transformation rules
- Commands to control rule applications
- Non-deterministic
- Computationally complete
- Formal semantics

Double-pushout graph transformation

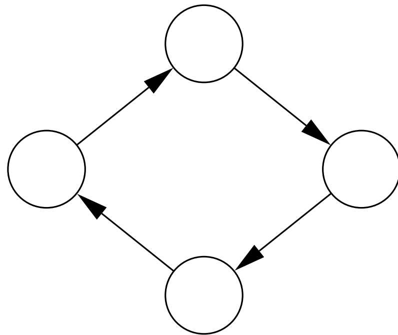


Double-pushout graph transformation

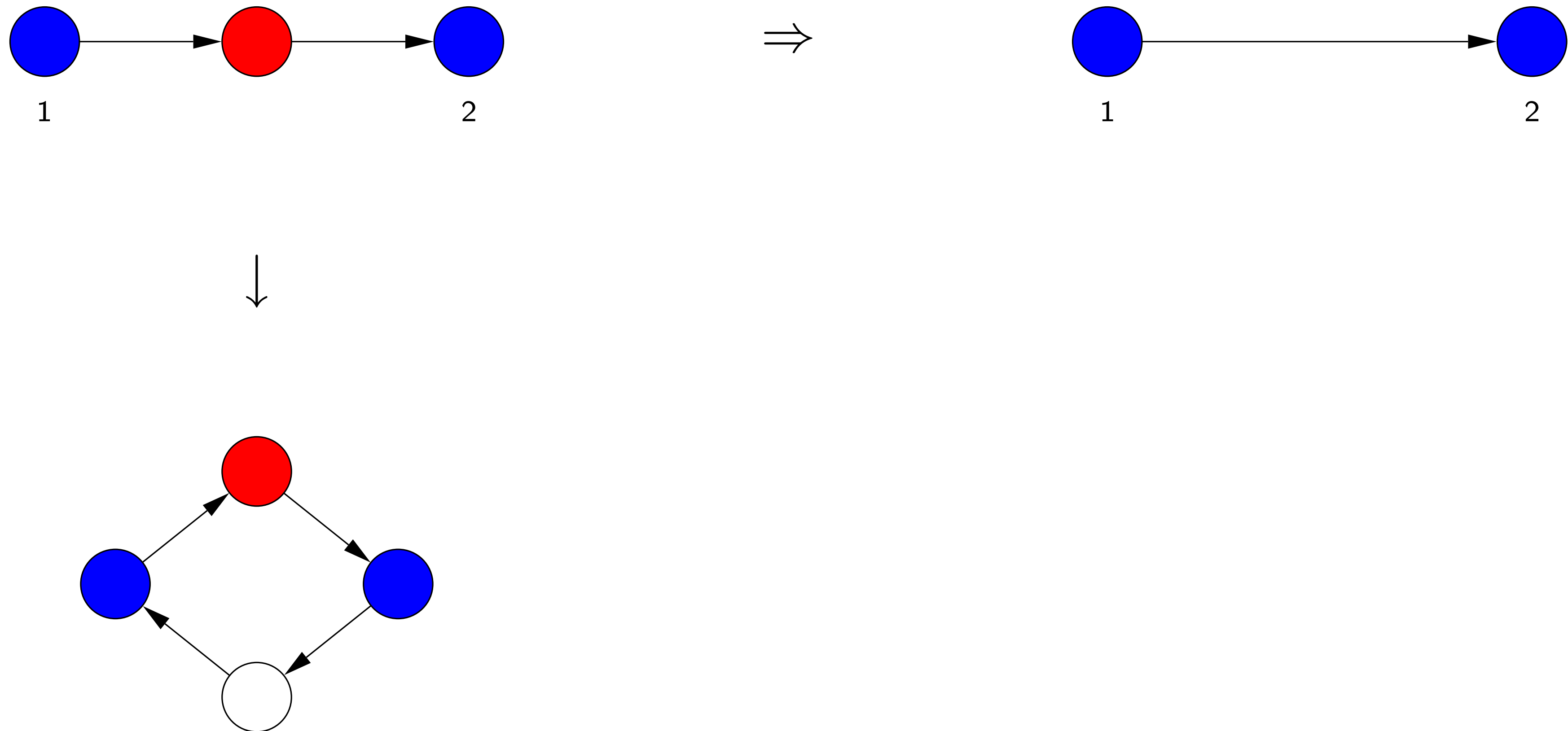


$\Rightarrow$

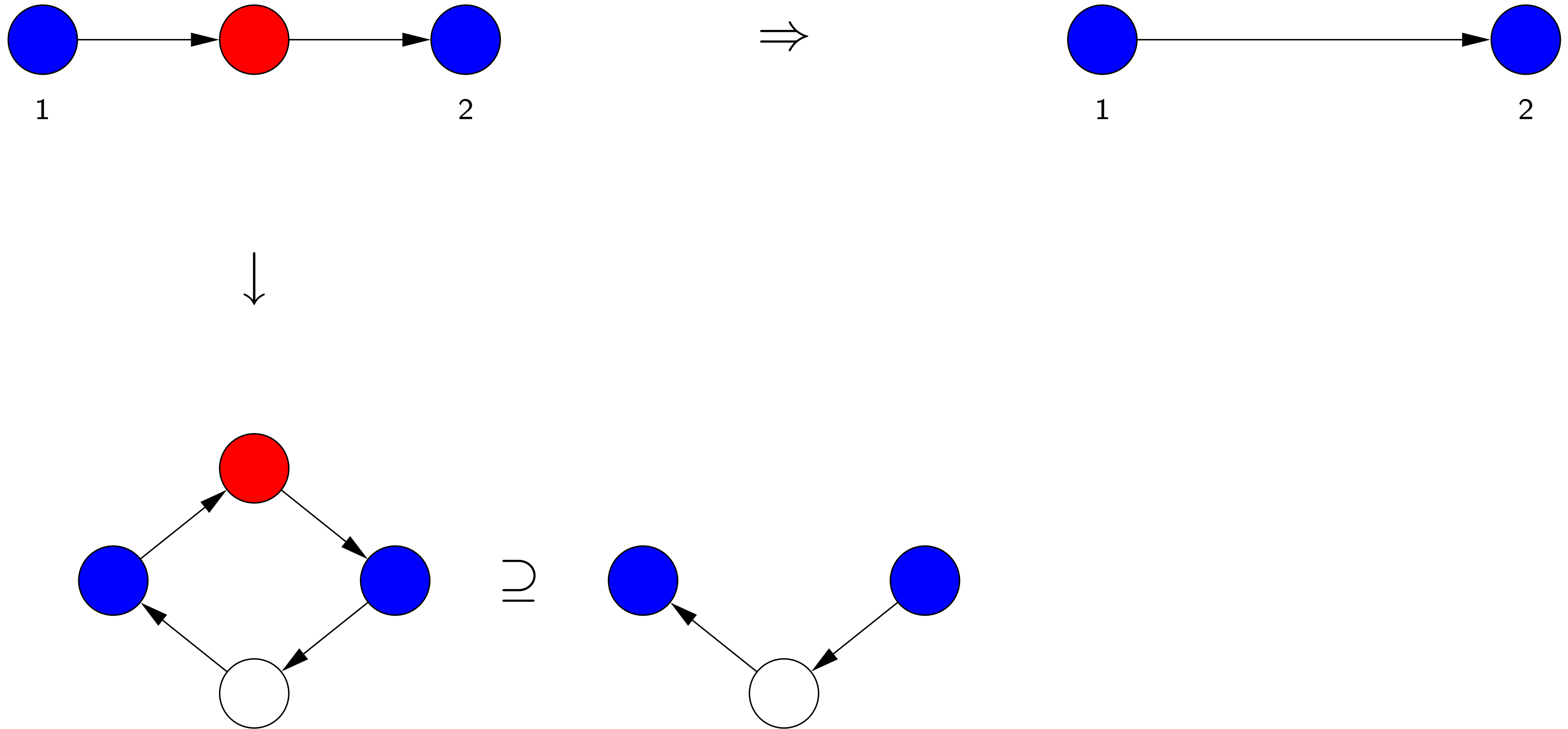
$\downarrow$



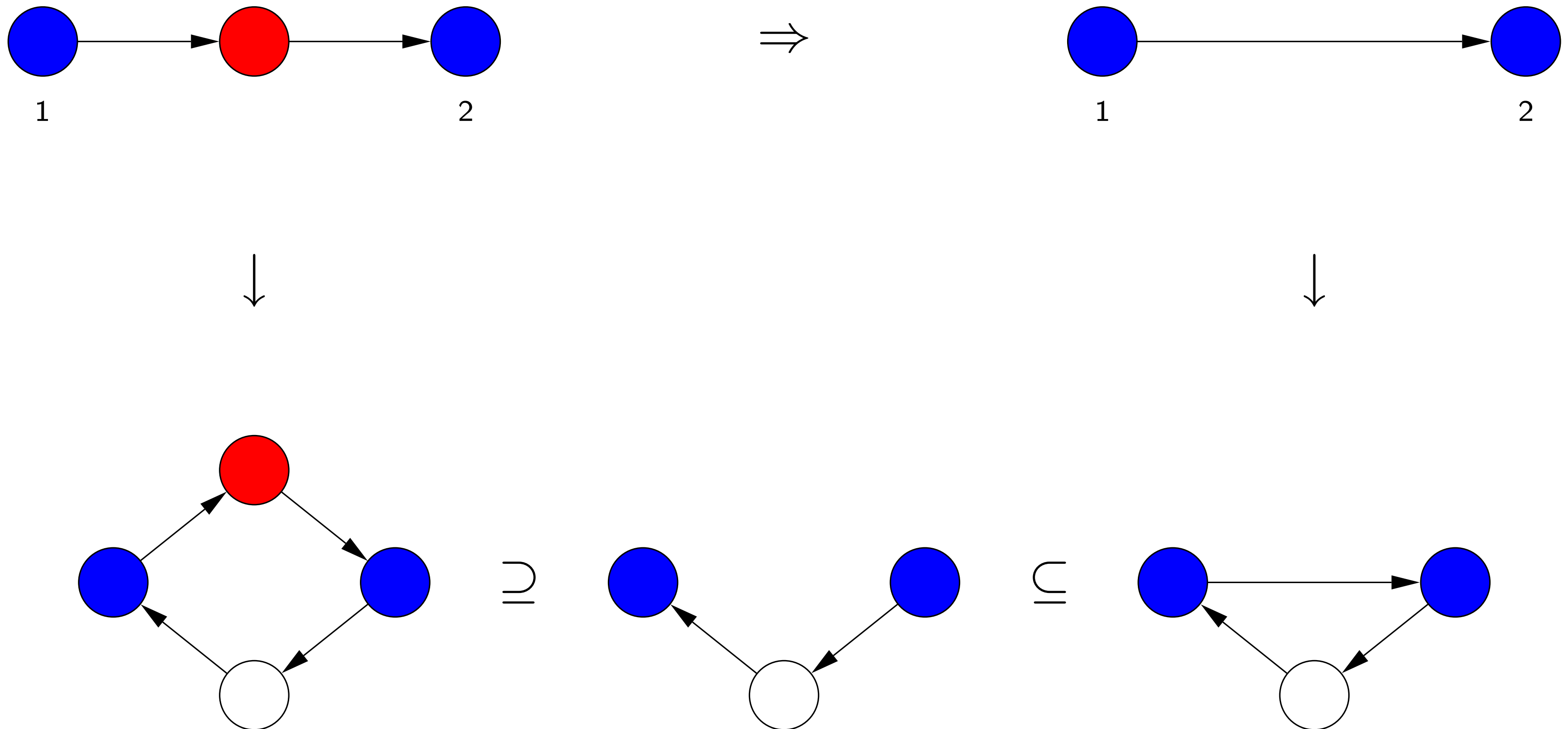
Double-pushout graph transformation



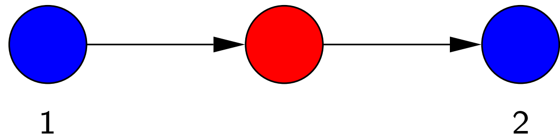
Double-pushout graph transformation



Double-pushout graph transformation

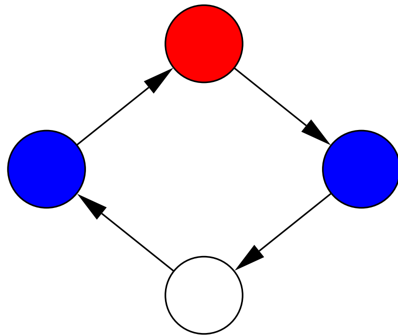


Double-pushout graph transformation

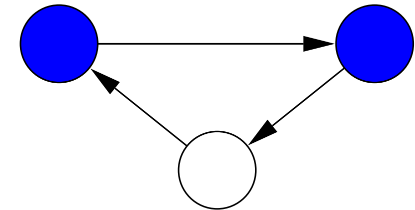


$\Rightarrow$

$\downarrow$ $\downarrow$

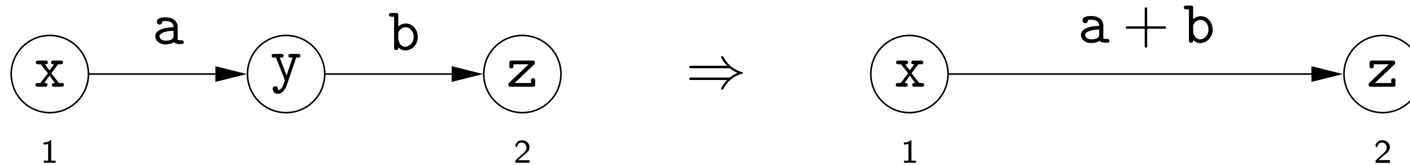


$\Rightarrow$



Rule schemata

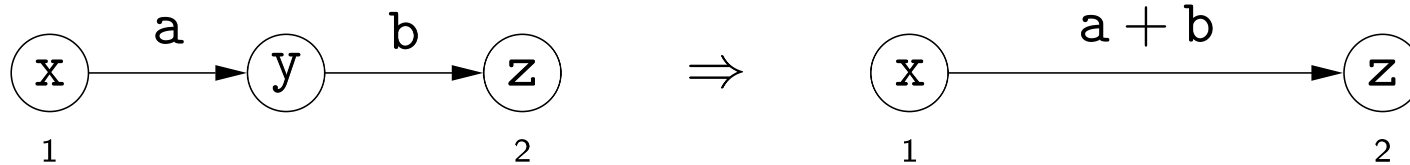
```
reduce (a,b,x,y,z:int)
```



- labels are (sequences of) expressions of type integer or string
- left-hand expressions are variables or constants
- right-hand expressions can contain arithmetic operators '+', '−', '∗' and '/'
- variables on the right side must also occur on the left side
- *relabelling* of interface nodes allowed

Rule schemata

```
reduce (a,b,x,y,z:int)
```

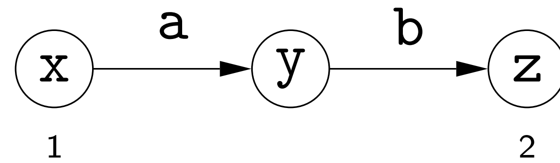


Application:

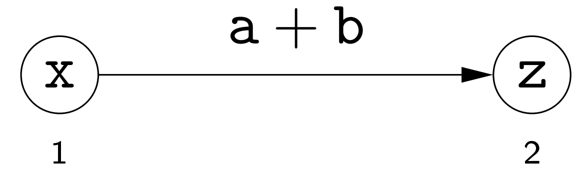
1. choose values for variables and evaluate expressions to obtain an *instance* of the rule schema
2. apply instance as normal

Rule schema application

Schema:

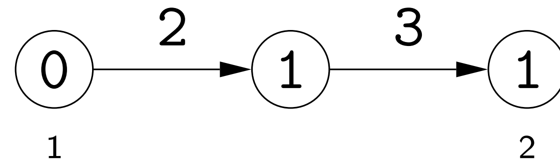


⇒



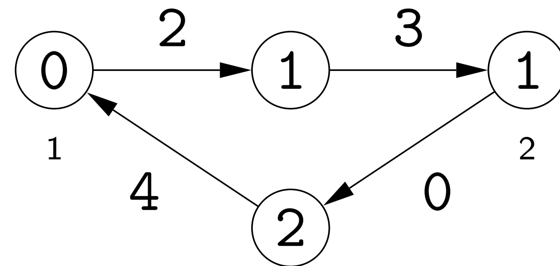
↓ ↓

Instance:

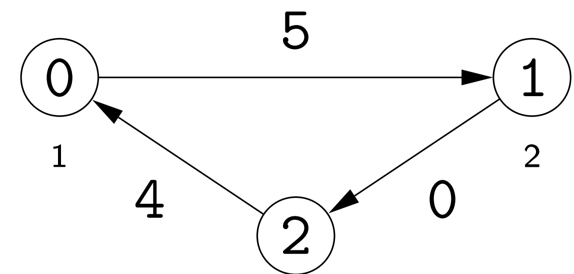


⇒

↓ ↓

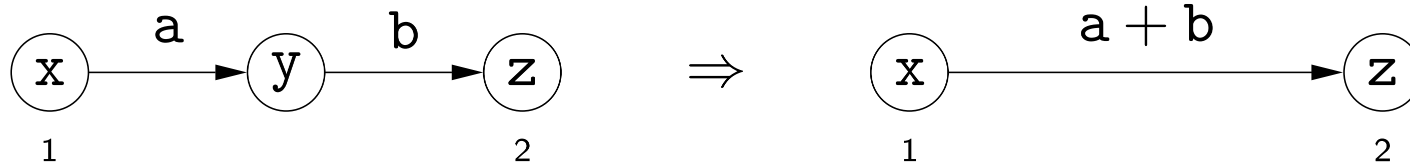


⇒



Conditional rule schemata

```
reduce (a,b,x,y,z:int)
```



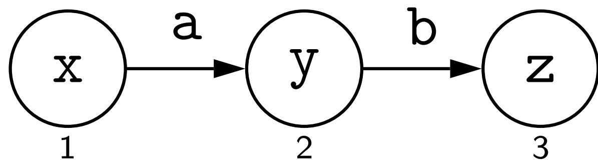
where $\mathtt{x} + \mathtt{a} + \mathtt{b} = \mathtt{z}$ and not $\mathtt{edge}(2, 1)$

- condition is a boolean expression built from arithmetic expressions, relational operators and logical operators
- variables in the condition must occur in the left graph
- special predicate $\mathtt{edge}(m, n)$ requires an edge from node m to node n (normally used in negated form)

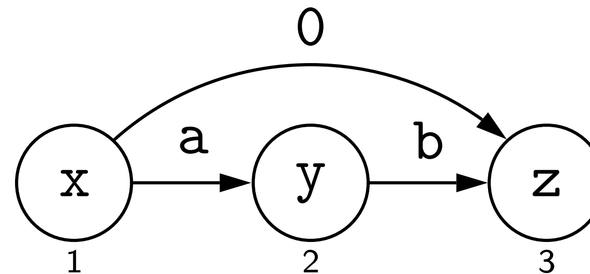
Example program: Transitive closure

main = link!

link(a, b, x, y, z : int)



$\Rightarrow$

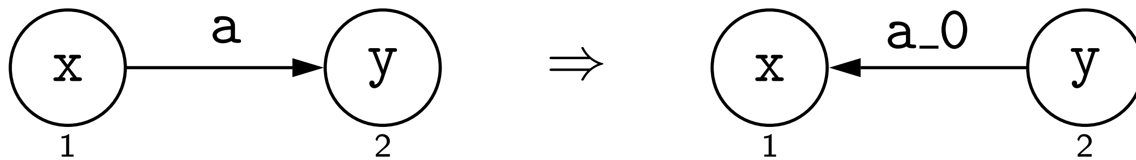


where not edge(1, 3)

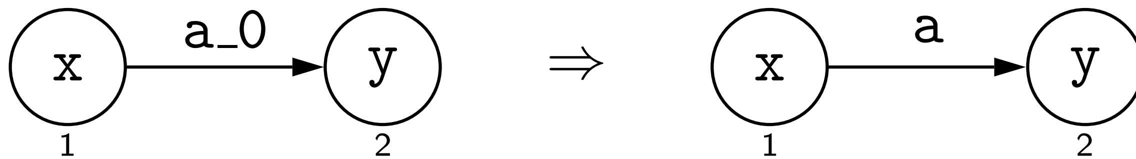
Example program: Inverse

```
main = reverse!; unmark!
```

`reverse`



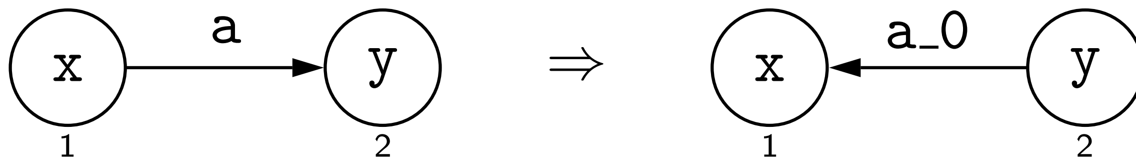
`unmark`



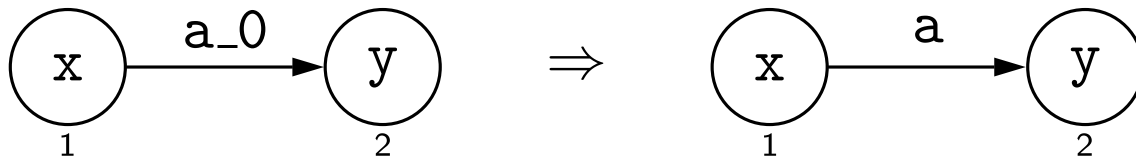
Example program: Inverse

`main = reverse!; unmark!`

`reverse`



`unmark`

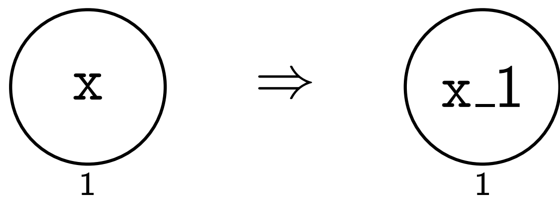


- *tagged label* $e_1_e_2_\ldots_e_n$ is a sequence of expressions

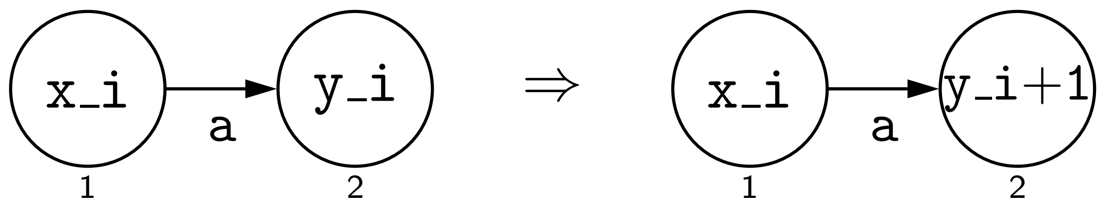
Example program: Vertex colouring

main = init!; {inc1, inc2}!

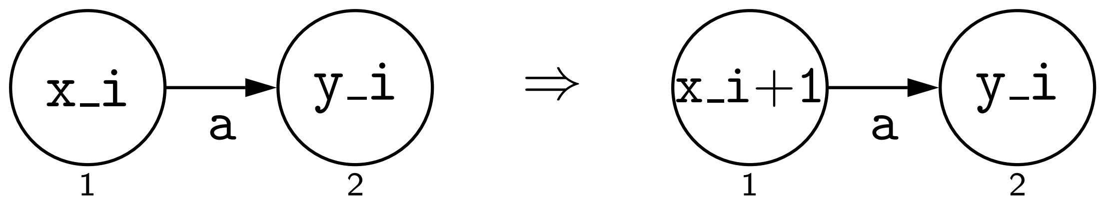
init



inc1



inc2



Example program: Vertex colouring (cont'd)

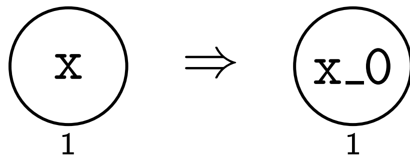
Proposition (Correctness)

For every integer-labelled graph G, the program terminates after $\mathrm{O}(|V_G|^2)$ rule applications and returns G correctly coloured.

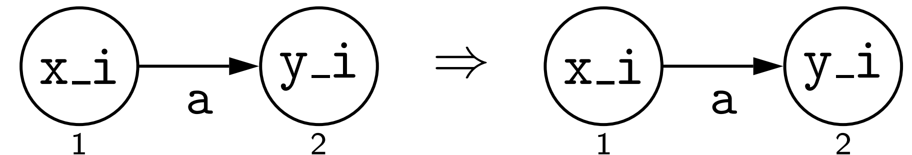
Example program: 2-colouring (testing for bipartiteness)

```
main = choose; colour!; if illegal then undo!
colour = {colour1, colour2}
```

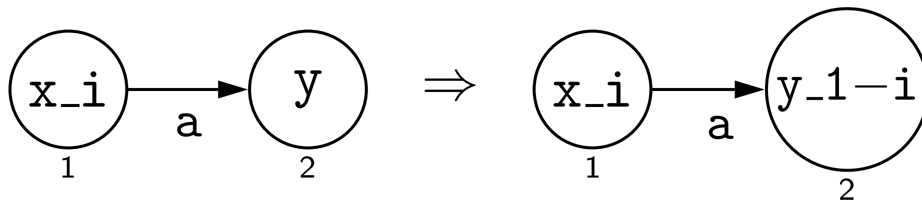
choose



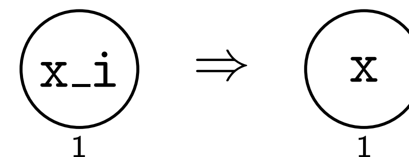
illegal



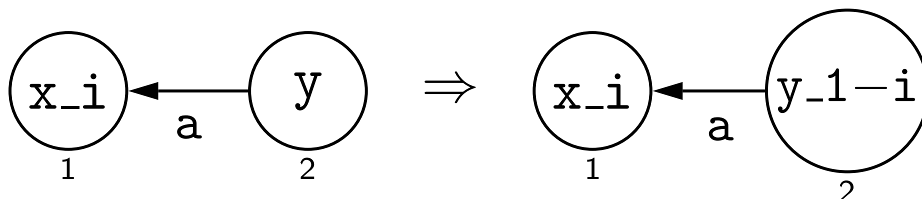
colour1



undo



colour2



Example program: 2-colouring (cont'd)

Proposition (Correctness)

For every integer-labelled graph G, the program terminates after $\mathrm{O}(|V_G|)$ rule applications. If G is not 2-colourable, then it is returned unmodified; otherwise it is returned 2-coloured.

GP abstract syntax

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | MacroDecl | MainDecl
MacroDecl	::=	MacroId '=' ComSeq
MainDecl	::=	`main` '=' ComSeq

ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | MacroCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| ComSeq '!'
		| `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
MacroCall	::=	MacroId

Structural operational semantics (SOS)

Small-step transition relation

$$\rightarrow \;\subseteq\; (\mathrm{ComSeq} \times \mathcal{G}) \times ((\mathrm{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\mathrm{fail}\})$$

defined by inference rules

- $\mathcal{G}$: set of all graphs labelled with integers and strings
- $(\mathrm{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\mathrm{fail}\}$: set of *configurations*

Notation on subsequent slides:

- P, P', Q, C: command sequences (programs)
- $\mathcal{R}$: rule-set call (set of conditional rule schemata)
- G, H: graphs

Rule application

$$[\mathrm{Call}_1]_{\mathrm{sos}} \qquad \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R},\ G \rangle \longrightarrow H}$$

$$[\mathrm{Call}_2]_{\mathrm{sos}} \qquad \frac{G \notin \mathrm{Dom}(\Rightarrow_{\mathcal{R}})}{\langle \mathcal{R},\ G \rangle \longrightarrow \mathsf{fail}}$$

Sequential composition

$$[\mathrm{Seq}_1]_{\mathrm{sos}} \qquad \frac{\langle P,\ G\rangle \longrightarrow \langle P',\ H\rangle}{\langle P;Q,\ G\rangle \longrightarrow \langle P';Q,\ H\rangle}$$

$$[\mathrm{Seq}_2]_{\mathrm{sos}} \qquad \frac{\langle P,\ G\rangle \longrightarrow H}{\langle P;Q,\ G\rangle \longrightarrow \langle Q,\ H\rangle}$$

$$[\mathrm{Seq}_3]_{\mathrm{sos}} \qquad \frac{\langle P,\ G\rangle \longrightarrow \mathsf{fail}}{\langle P;Q,\ G\rangle \longrightarrow \mathsf{fail}}$$

Branching

$$[\mathrm{If}_1]_{\mathrm{sos}} \qquad \frac{\langle C,\ G\rangle \longrightarrow^{+} H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \longrightarrow \langle P,\ G\rangle}$$

$$[\mathrm{If}_2]_{\mathrm{sos}} \qquad \frac{C \text{ finitely fails on } G}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\ G\rangle \longrightarrow \langle Q,\ G\rangle}$$

Finite failure

Program C *finitely fails* on graph G if the following holds:

(1) Termination: there is no infinite transition sequence $\langle C,\, G\rangle \rightarrow \langle C_1,\, G_1\rangle \rightarrow \langle C_2,\, G_2\rangle \rightarrow \ldots$

(2) Failure: for every terminal configuration[1] γ, $\langle C,\, G\rangle \rightarrow^* \gamma$ implies $\gamma = \text{fail}$.

[1] i.e. there is no γ' such that $\gamma \rightarrow \gamma'$

Finite failure

Program C *finitely fails* on graph G if the following holds:

(1) Termination: there is no infinite transition sequence $(C, G) \to (C_1, G_1) \to (C_2, G_2) \to \dots$

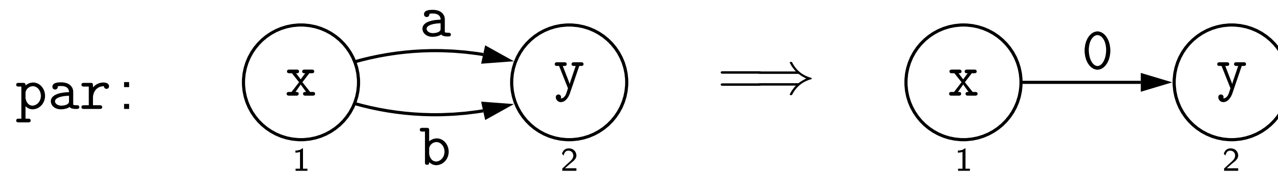
(2) Failure: for every terminal configuration[1] γ, $\langle C, G \rangle \to^* \gamma$ implies $\gamma = \text{fail}$.

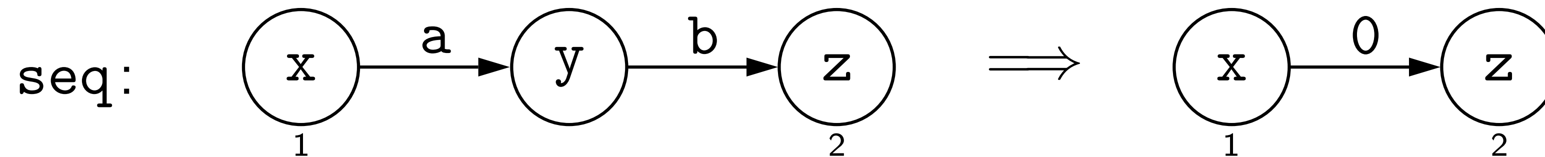
Similar to *negation as failure* in logic programming [Clark 78]

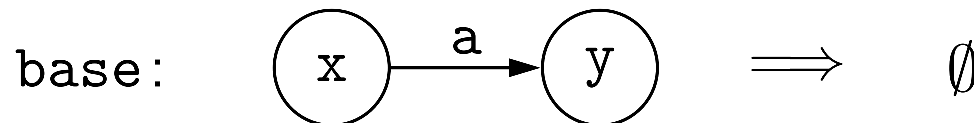
[1] i.e. there is no γ' such that $\gamma \to \gamma'$

Example: Testing destructively for series-parallel graphs

$\texttt{main} = \texttt{if}\ \{\texttt{par}, \texttt{seq}\}!;\ \texttt{base}\ \texttt{then}\ P\ \texttt{else}\ Q$







assuming a connected input graph

Iteration

$$[\mathrm{Alap}_1]_{\mathrm{sos}} \qquad \frac{\langle P,\ G\rangle \to^{+} H}{\langle P!,\ G\rangle \to \langle P!,\ H\rangle}$$

$$[\mathrm{Alap}_2]_{\mathrm{sos}} \qquad \frac{P \text{ finitely fails on } G}{\langle P!,\ G\rangle \to G}$$

Iteration

$$[\mathrm{Alap}_1]_{\mathrm{sos}} \qquad \frac{\langle P,\ G\rangle \to^{+} H}{\langle P!,\ G\rangle \to \langle P!,\ H\rangle}$$

$$[\mathrm{Alap}_2]_{\mathrm{sos}} \qquad \frac{P \text{ finitely fails on } G}{\langle P!,\ G\rangle \to G}$$

Note: P is an arbitrary program, allowing nested loops

Derived constructs

$[\text{Skip}]_{\text{sos}}$ $\langle \texttt{skip}, G \rangle \to \langle \texttt{r}, G \rangle$

where $\texttt{r}$ is an identifier for $\emptyset \Rightarrow \emptyset$

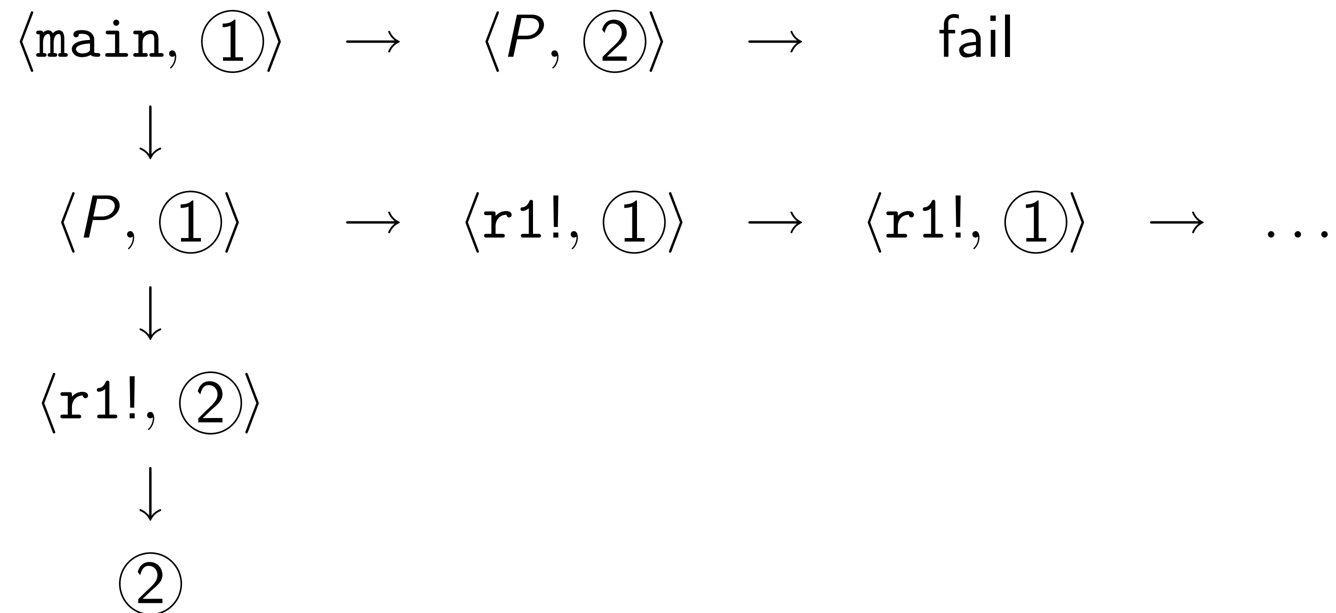
$[\text{Fail}]_{\text{sos}}$ $\langle \texttt{fail}, G \rangle \to \langle \{\}, G \rangle$

$[\text{If}_3]_{\text{sos}}$ $\langle \texttt{if}\ C\ \texttt{then}\ P, G \rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip}, G \rangle$

Example: Nondeterminism

$\texttt{main} = \{\texttt{r1}, \texttt{r2}\};\ \{\texttt{r1}, \texttt{r2}\};\ \texttt{r1}!$

$\texttt{r1}: ① \Rightarrow ① \qquad \texttt{r2}: ① \Rightarrow ②$



where $P = \{\texttt{r1}, \texttt{r2}\};\ \texttt{r1}!$

Semantic function $[\![_]\!]_{\mathrm{sos}}$

$[\![_]\!]_{\mathrm{sos}} : \mathrm{ComSeq} \longrightarrow (\mathcal{G} \longrightarrow 2^{\mathcal{G} \cup \{\perp\}})$ is defined by

$$
\begin{array}{rcl}
[\![P]\!]_{\mathrm{sos}} G & = & \{H \in \mathcal{G} \mid \langle P,\, G\rangle \xrightarrow{+} H\} \cup \\
 & & \{\perp \mid P \text{ can diverge or get stuck from } G\}
\end{array}
$$

where

- P *can diverge from* G if there is an infinite sequence $\langle P,\, G\rangle \longrightarrow \langle P_1,\, G_1\rangle \longrightarrow \langle P_2,\, G_2\rangle \longrightarrow \ldots$
- P *can get stuck from* G if there is a terminal configuration $\langle Q,\, H\rangle$ such that $\langle P,\, G\rangle \longrightarrow^* \langle Q,\, H\rangle$.

Semantic equivalence

Programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!]_{\mathrm{sos}} = [\![Q]\!]_{\mathrm{sos}}$

Semantic equivalence

Programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!]_{\mathrm{sos}} = [\![Q]\!]_{\mathrm{sos}}$

Examples

- $P; \texttt{skip} \;\equiv\; P \;\equiv\; \texttt{skip}; P$
- $\texttt{fail}; P \;\equiv\; \texttt{fail}$
- $\texttt{if}\ C\ \texttt{then}\ (P_1; Q)\ \texttt{else}\ (P_2; Q) \;\equiv\; (\texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2); Q$
- $P! \;\equiv\; \texttt{if}\ P\ \texttt{then}\ (P; P!)$

Counterexample

- $P; \texttt{fail} \;\not\equiv\; \texttt{fail}$

Bounded nondeterminism

Lemma

For every program P and graph G, if $[\![P]\!]_{\mathrm{SOS}}G$ is infinite then P can diverge from G.

Bounded nondeterminism

Lemma

For every program P and graph G, if $[\![P]\!]_{\mathrm{SOS}} G$ is infinite then P can diverge from G.

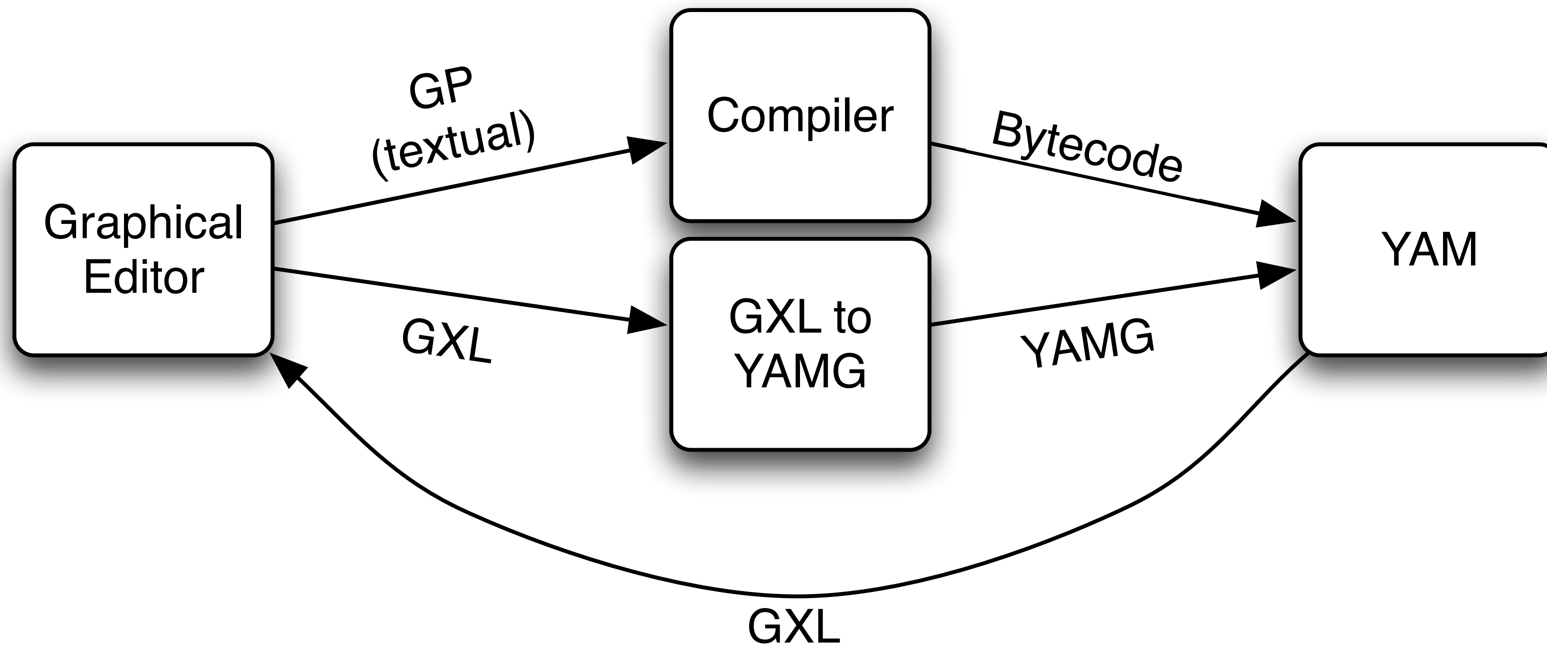
Example

$\texttt{main} = \{\texttt{stop}, \texttt{continue}\}!$

$\texttt{stop}$	$\texttt{continue}$
$① \Rightarrow \emptyset$	$① \Rightarrow ①\,②$

$[\![\texttt{main}]\!]_{\mathrm{SOS}}\, ① = \{\bot, \emptyset, ②, ②\,②, ②\,②\,②, \ldots\}$

GP Programming System



Graphical editor

Graph Programs

File Help

Graphs

UnnamedGraph

Programs

sierpinski

Program Editor | Graph Editor | Execution | Results | Options

init
inc
expand

Rule Name: expand

Na...	Type
x	INT
y	INT
a	INT
b	INT

New Rule

Duplicate

Layout | Bind new no... | next Label: 3 | Auto Increment

All Matches where:

Program Text

```
main = init ; (inc; expand!)! .
```

York abstract machine (YAM)

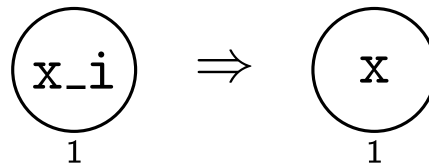
- Similar to Warren abstract machine for Prolog
- Written in C, using Judy dynamic arrays
  - Input: YAM bytecode and graphs
  - Output: (sets of) graphs
- Complex data structure for current graph to permit fast queries
- Executes low-level operations on graphs needed for graph matching and transformation, e.g. "Find nodes with label *l*"
- Maintains a change/choice stack, to allow backtracking

GP compiler

- Translates textual GP to YAM bytecode
- Written in Haskell
- Compiles individual rules (producing code for graph matching) and composes them according to GP's control constructs
- For each rule, generates a searchplan for graph matching
- Joins rules into an overall program using code concatenation for "`;`", and functions for "`!`" and "`if-then-else`"

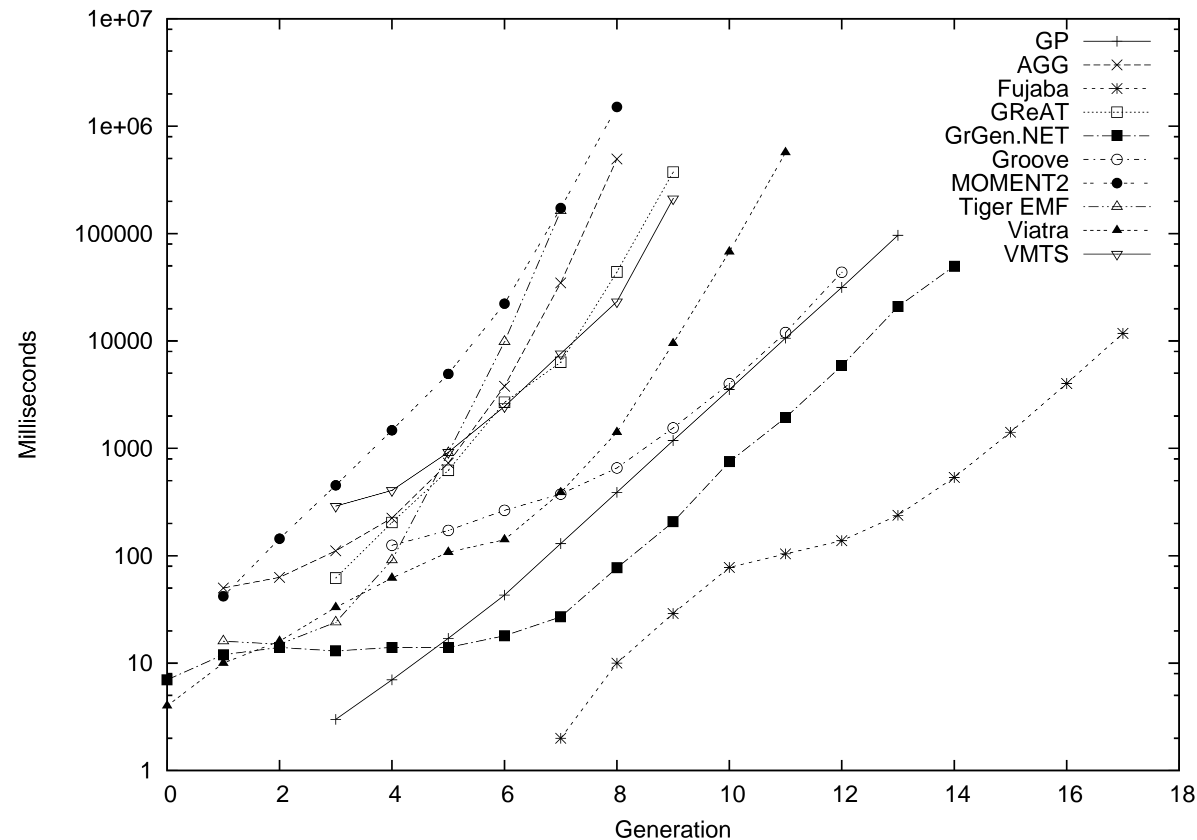
Example: Compilation of undo

undo(i, x: int)



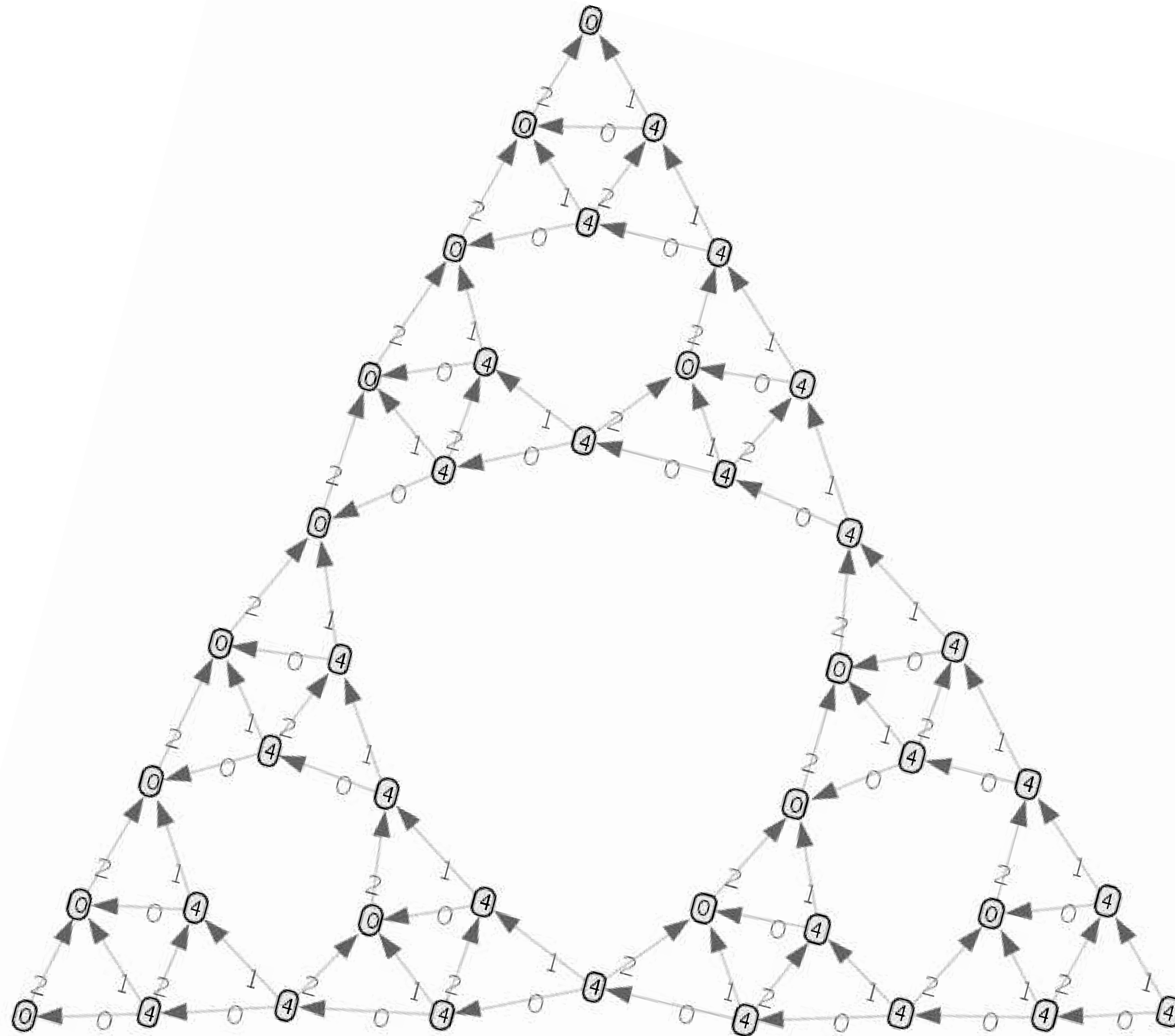

```
.func undo(0/0)
0 PushI 2
1 LNodesWithLabelSize
2 PushI 0
3 PushI 0
4 LNodesWithType
5 PushI 0
6 PushI 1
7 LNodesWithType
8 Call "PickFromList3"
9 Drop_label
10 Succ
```

Sierpinski benchmark



Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools, Proc. AGTIVE 2007, LNCS

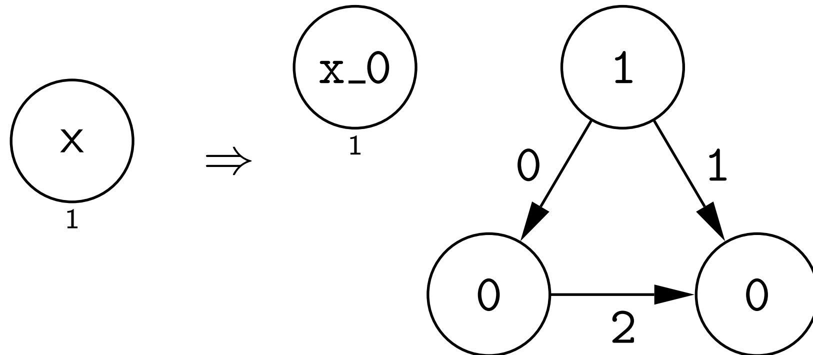
Sierpinski triangle (3rd generation)



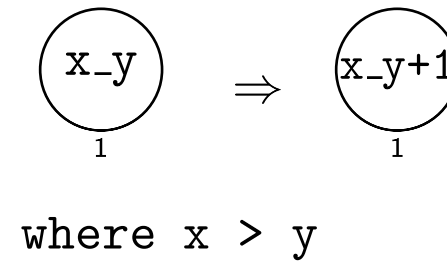
Generating Sierpinski triangles

main = init; (inc; expand!)!

init

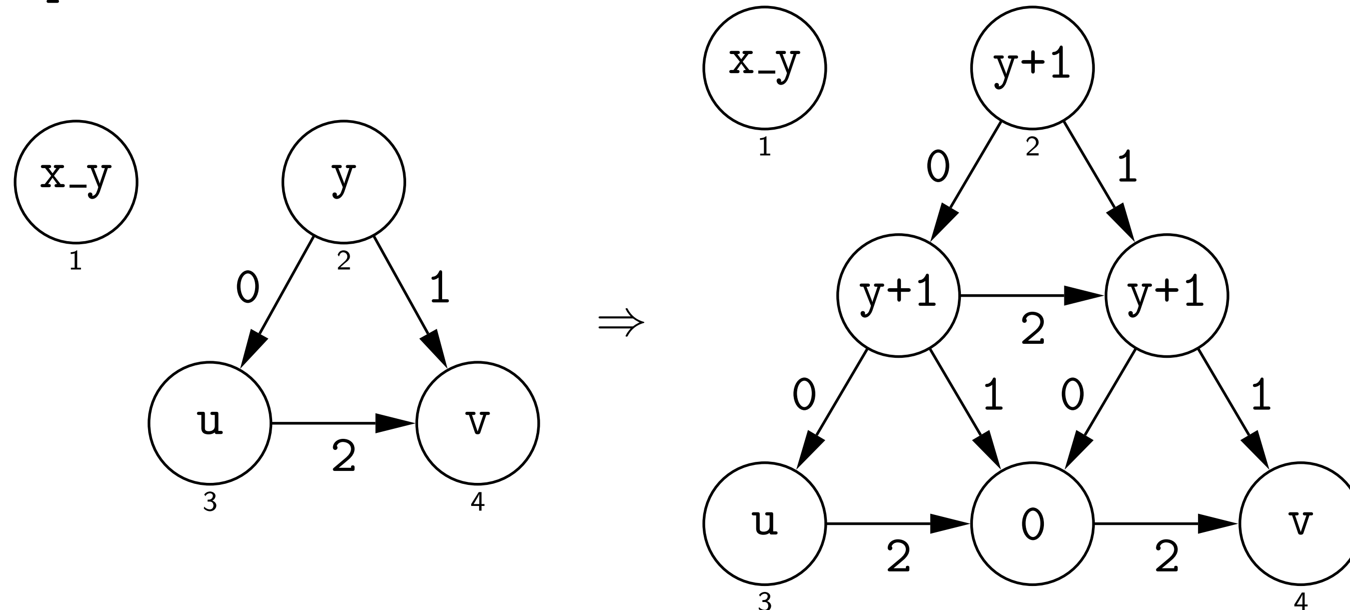


inc



where x > y

expand



Conclusion

- High-level language for solving graph problems
- Rule-based, visual programming
- Simple syntax and semantics facilitate formal reasoning
- Alternative formal semantics: operational (SOS) and denotational (least fixed-point)
- Implementation: reasonably fast, faithful to the semantics, complete for terminating programs

Future work

- Procedures: see Sandra Steinert's thesis
- Typing: restricting the shape of graphs
- Static analysis: checking for termination and confluence
- Program verification: calculus and tool support