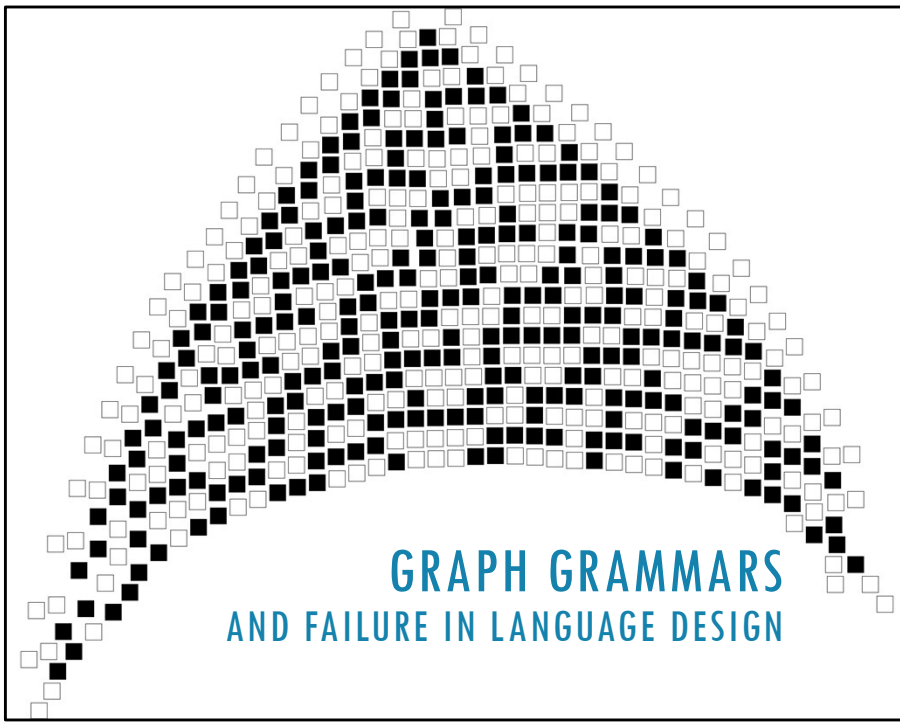


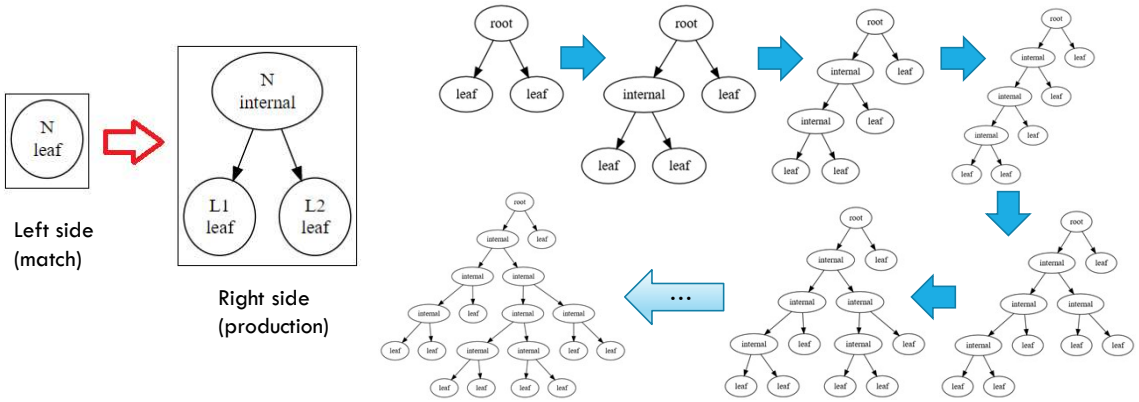
GRAPH GRAMMARS

AND FAILURE IN LANGUAGE DESIGN

Mark Gritter (he/him)
Twitter: @markgritter
Minnebar 15

WHAT'S A GRAPH GRAMMAR?

Here's one! (You can play with it at http://soffit.combinatorium.com/)



Left side (match)

Right side (production)

A graph grammar consists of a set of graph rewriting rules. It's like a context-sensitive grammar, but it operates on graphs instead of strings.

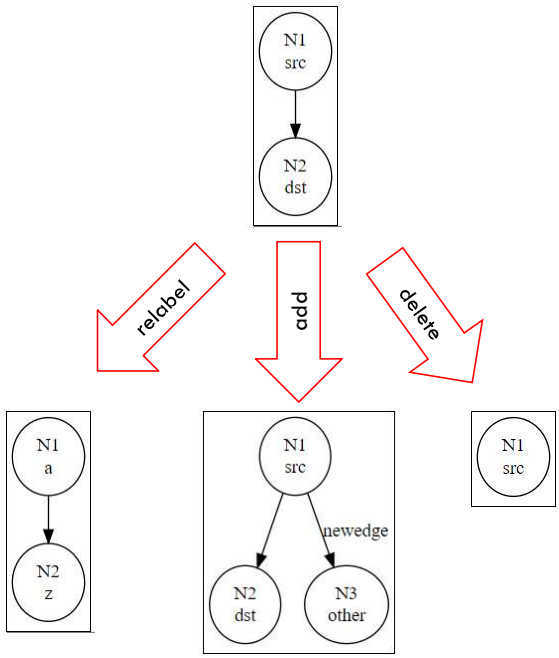
The left hand side is a graph to match. The right hand side shows how to transform that graph. In this example, we find a node labelled "leaf" and change its label to "internal", then add two more leaves. You can see 12 iterations of this rule on the right.

GRAPH GRAMMAR

aka "Graph Rewriting System"

A set of production rules for labeled graphs

- Match some subgraph
- Apply a transformation



More formally, a graph grammar or "graph rewriting system" is a set of production rules for labeled graphs. We can relabel the nodes and edges, add components, delete them, or merge nodes.

So what are they useful for?

VISUAL LANGUAGES

A syntax for visual languages!

Grammar from:
J. Rekers and A. Schurr. 1995.
A graph grammar approach to graphical parsing.
In *Proceedings of the 11th International IEEE Symposium on Visual Languages (VL '95)*. IEEE Computer Society, USA, 195.

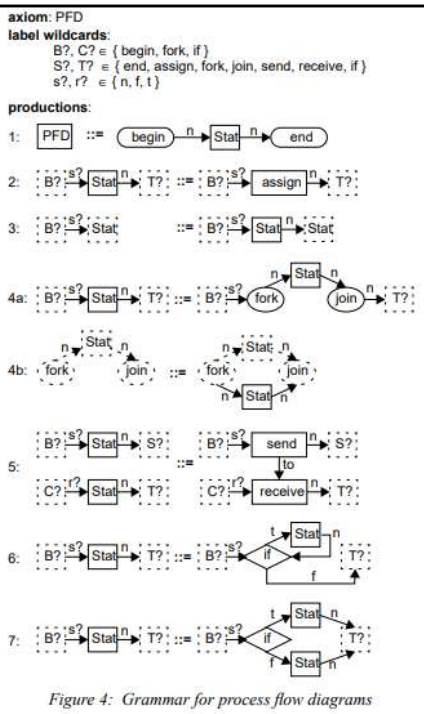


Figure 4: Grammar for process flow diagrams

Graph grammars can be a syntax for a visual language, just like context-free grammars provide the syntax for textual languages.

. . . LIKE UML!

Frank Hermann, 2006. **A typed attributed graph grammar for syntax-directed editing of UML sequence diagrams.**

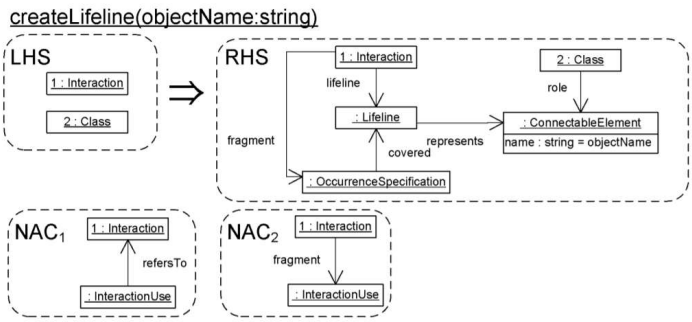


Figure 5.3: Rule createLifeline()

Here's an example graph transformation that helps edit UML diagrams.

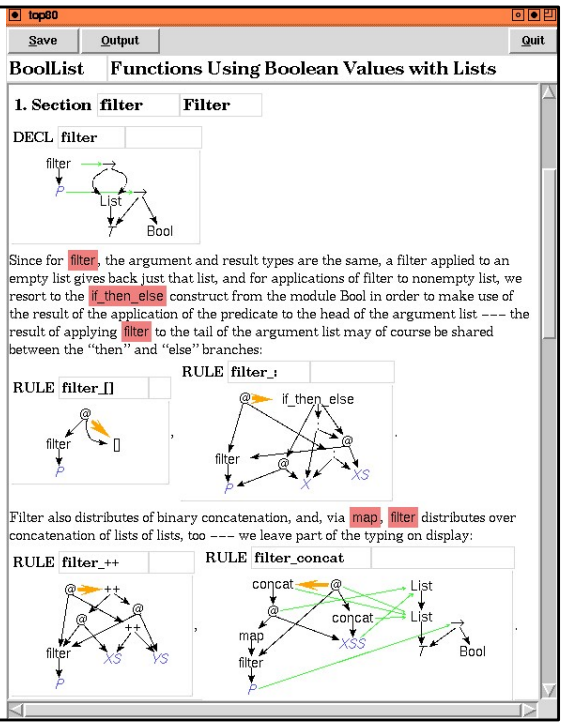
(The graph grammar Dr. Hermann used in his PhD thesis is somewhat more complicated, it has "negative acceptance criteria", features that should not be present.)

CODE ANALYSIS AND REWRITING

http://www.cas.mcmaster.ca/~kahl/HOPS/

HOPS is intended as:

- **Research tool:** New languages can easily be constructed and experimented with via the transformation mechanism.
- **Programming tool:** HOPS may be used as a ``better Haskell editor'', integrating syntax directed editing with strong online typing.
- **Visualisation and debugging tool:** Automatic evaluation sequences help to illustrate the workings of, for example, purely functional programs with lazy evaluation.



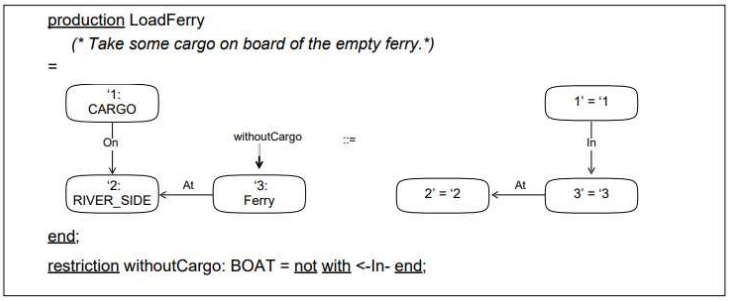
Even textual computer code can be understood as an abstract syntax tree, which is a type of graph, or as a "term graph". Here's an image from HOPS, a system for working with term graphs that provides the capabilities to transform code and visualize it.

A MODEL OF CONCURRENT OR NONDETERMINSTIC PROGRAMMING

Graph grammar productions can operate on any match found, and in any order.

Albert Ziindorf, I., & Schiirr, A. (1991). **Nondeterministic Control Structures for Graph Rewriting Systems.**

Figure 3: The production LoadFerry



Many graph grammars allow nondeterminstic execution; any rule that matches may be applied. So they're an interesting model for concurrent or randomized execution. Shown here is an example production rule that's part of the sheep-wolf-cabbage problem.

CREATING NETWORKS

Hartmut Ehrig, Annegret Habel, and Hans-Jörg Kreowski. **Introduction to graph grammars with applications to semantic networks.** *Computers & Mathematics with Applications*, 23(6-9):557–572, 1992

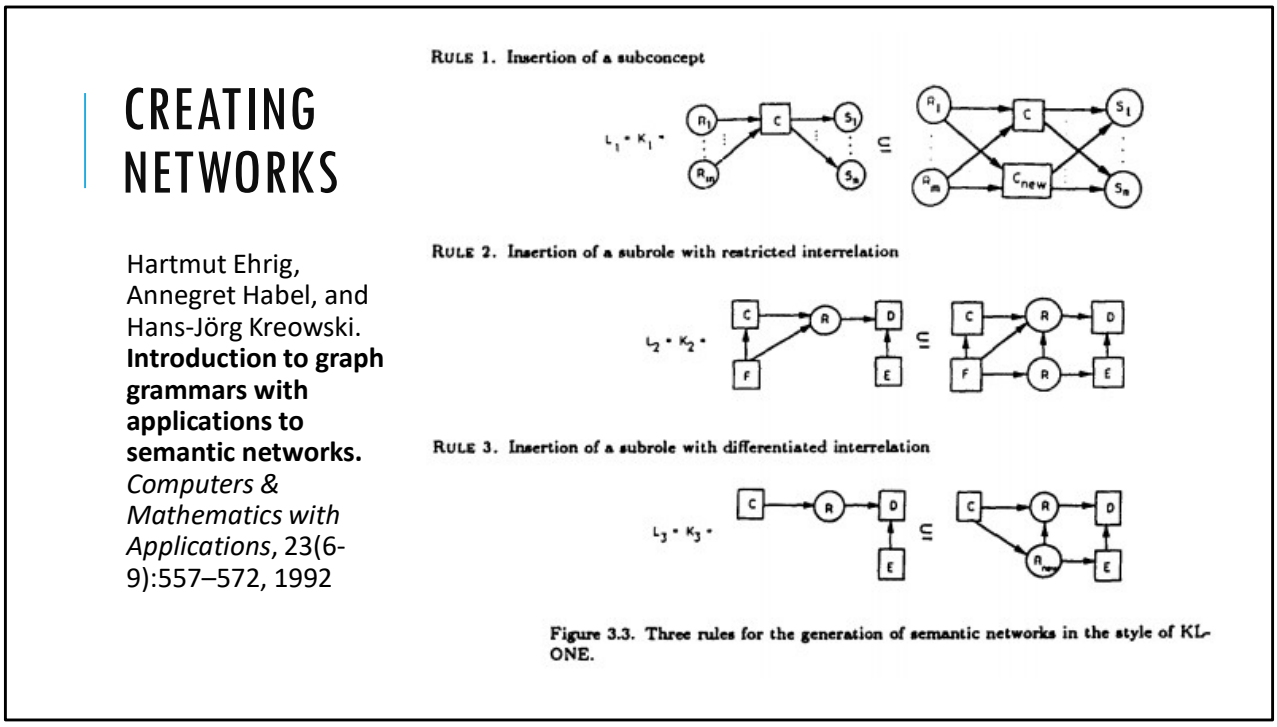


Figure 3.3. Three rules for the generation of semantic networks in the style of KL-ONE.

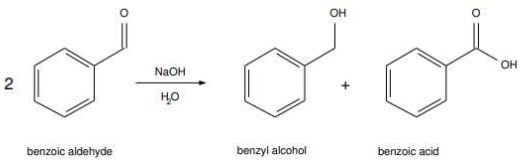
The side of things I'm most excited about is their ability to create graphs, and other content that can be represented as graphs! Here's an example of rules to produce a semantic network.

CREATING MOLECULES

https://www.tbi.univie.ac.at/software/GGL/

"Graph Grammar Library":

Beside its general applicability, it features an extensive chemistry module to handle grammar-based graph transformation in Chemistry. **Since a chemical reaction can be viewed as graph transformation** from the set of educt graphs to the set of product graphs, the GGL can be used to iteratively expand the topology of chemical reaction network, starting from a set of chemical reactions (graph rewrite rules) and set of molecules (vertex and edge labeled graphs) using the general DPO framework of the GGL.




```
rule [
  ruleID "cannizzaro reaction too general"
  context [
    node [ id 1 label "C" ]
    node [ id 2 label "O" ]
    node [ id 3 label "H" ]
    node [ id 4 label "O" ]
    node [ id 5 label "C" ]
    node [ id 6 label "H" ]
    node [ id 7 label "H" ]
    node [ id 8 label "H" ]
    node [ id 9 label "O" ]
    edge [ source 1 target 7 label "-" ]
    edge [ source 4 target 8 label "-" ]
    edge [ source 5 target 9 label "=" ]
  ]
  left [
    edge [ source 1 target 2 label "=" ]
    edge [ source 3 target 4 label "-" ]
    edge [ source 5 target 6 label "-"
  ]
  right [
    edge [ source 1 target 2 label "-" ]
    edge [ source 2 target 3 label "-" ]
    edge [ source 4 target 5 label "-" ]
    edge [ source 6 target 1 label "-" ]
  ]
]
```

Molecules can be represented as graphs, so chemical reactions are graph transformations! Here's one on the right, in a syntax even worse than mine.

CREATING DUNGEONS

Joris Dormans and S. Bakkes, "**Generating Missions and Spaces for Adaptable Play Experiences**," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 216-228, Sept. 2011.

"Cyclic Dungeon Generation": https://www.youtube.com/watch?v=mA6PacEZX9M, the basis behind the game "Unexplored"

Fig. 9. Rule to move a lock forward. The circular node marked with a question mark indicates any node. Applying this rule will not change the type of this node, only its location and connections in the graph.

Fig. 10. Rule to move a key backwards by relocating a task from behind the door to the position right in front of the key.

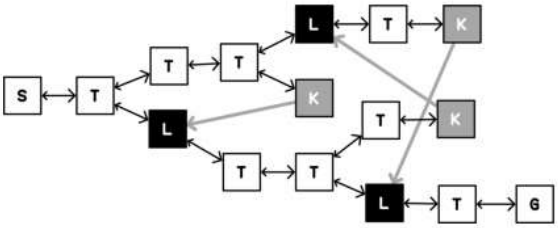


Fig. 11. Reorganized mission with keys and locks.

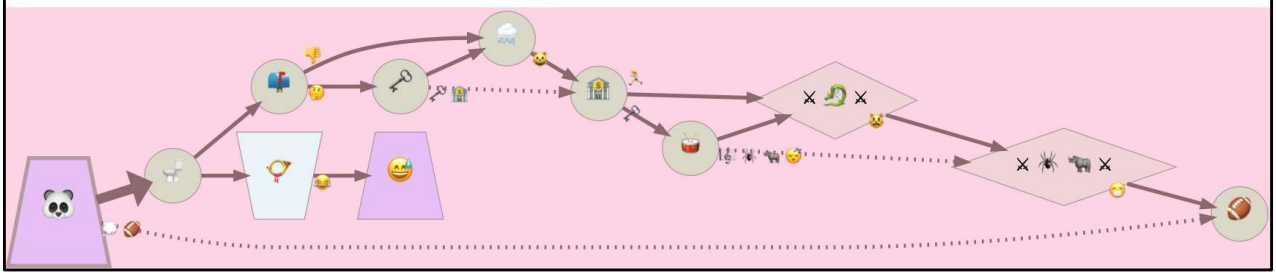
One recent use of graph grammars is in the game "Unexplored". Joris Dormans published work on creating missions and spaces with graph grammars, then applied it in the design of the game. He didn't publish direction on "Unexplored" but the YouTube video from ProcJam is very good.

CREATING ADVENTURES

https://twitter.com/tinyperil by Ian Holmes

Built with: https://github.com/ihh/graphgram

"The epic Thrash metal of the Red and the disgusting construction worker"



A more recent effort along those lines is the twitter bot "Tiny Peril" which creates adventure-like graphs. Ian Holmes wrote a small Javascript graph grammar engine that's the basis of this work.

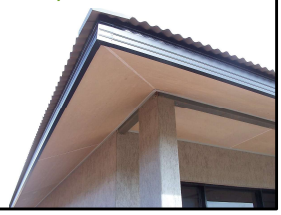
GREAT! HOW DO I GET STARTED?

There are several very good implementations, but it turned out I hated them all.

- Large integrated environment (like PROGRES) or big libraries with complicated syntax.
- How would I *ever* manage to embed one in a small generative art project, like the MetaArtBot? http://artbot.combinatorium.com/

Inspiration: Tracery: https://tracery.io/

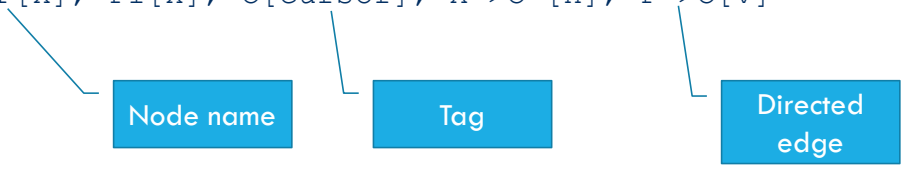
- So I picked the name "Soffit"



SOFFIT

I already knew Graphviz's Dot language, so graphs in Soffit are written in Dot syntax, sort of.

```
X[x]; P[x]; P1[x]; C[cursor]; X->C [h]; P->C[v]
```



A graph in soffit has named vertices, with tags attached to vertices or edges.

SOFFIT [2]

Graph grammars are JSON objects:

```
{
  "version": "0.1",
  "X--Y [?]": [
    "X--Y",
    "X; Y"
  ],
  "start": "A--B--C--D--E--A--D--B--E--C--A [?]"
}
```

Following Tracery's design, graph grammars in Soffit are JSON objects with the left hand side as the key, and the right hand side is the value (or a list of values, each of which could be used.) Here's a simple Soffit grammar that generates a random graph with 5 vertices.

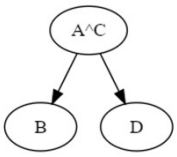
SOFFIT EXTENSIONS TO DOT SYNTAX

Bidirectional arrows and chaining

```
"A->B->C<-D [tag]": "A->B; C<-D;"
```

Merge two nodes

```
"A->B [tag]; C->D [tag]": "A^C; A->B; C->D"
```



Unicode/Emoji as tag and node names

```
"六->七 [☺]"
```

Graphviz integration: attributes are rendered without quoting

```
“X[color=red; style=filled]"
```

There are a few additions to the Dot syntax. One is that edges can be chained together, or written in the opposite direction.

The caret character, standing for “join” specifies that two vertices should be merged together in the right-hand side of a rule.

Like Swift, nearly any Unicode character is a valid identifier.

SOFFIT'S NONDETERMINISM

1. Shuffle all the rules
2. Try each rule in this random order
   - If it has multiple right-hand sides, shuffle them too
3. If there's a valid match, pick randomly from all valid matches (up to 1000)

SOFFIT IMPLEMENTATION

Soffit is implemented in Python

https://github.com/mgritter/soffit

Command-line tools:

- Execute a graph grammar (or sequence of them) and plot the result as an SVG
- Show a rule set as an HTML page

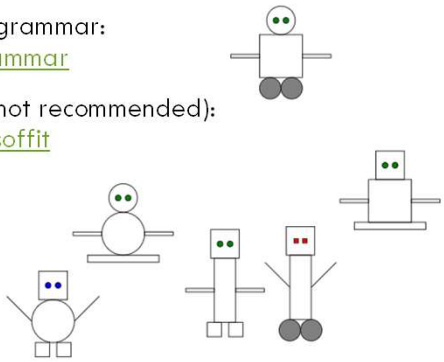
Online version built later, with AWS Lambda.

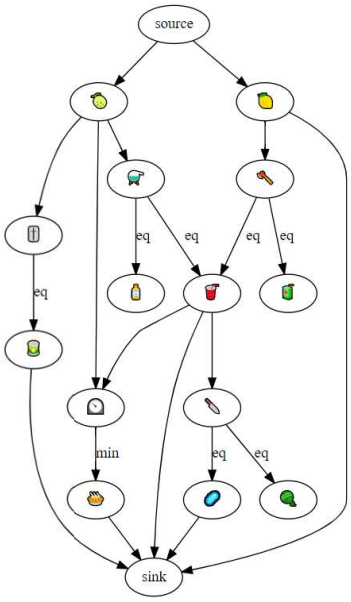
PROJECTS

Emojiconomy: github.com/mgritter/emojiconomy
- See my Roguelike Celebration 2020 talk!

SVG output from a Soffit grammar:
github.com/mgritter/svgrammar

Advent of Code in Soffit (not recommended):
github.com/mgritter/aoc-soffit





So, I was successful at using Soffit in other projects! I built a system called Emojiconomy that builds toy economies, which I talked about at this year's Roguelike Celebration last week.
I'm working on an extension that produces SVG output, shown here creating some small robots.
As a stunt, I tried to use Soffit to solve Advent of Code problems last year and I did manage to solve a few, but very slowly because the graphs get large.

MISTAKES

JSON sucks.

- No comments! (at least, not with Python's default parser)
- Constant comma problems
- No newlines! Very difficult for large rules
- I didn't write the implementation in JavaScript anyway

Left and Right usually share a lot

- Other tools have a separate "context" section

No templating for tags

- Dozens of versions of the same rule

Also, no way to name rules for debugging

EXAMPLE

```
    "COMMENT[these rules don't have a stopping point. They also blow away some next_char
arrows, which may be a bug.]" : "",

    "X[token]; Y[token]; XX[letter]; YY[letter]; X->XX; Y->YY; XX->XX [0]; YY->YY [0]" :
    "X[token]; Y[token]; XX[letter]; YY[letter]; X->XX; Y->YY; XX->XX [0]; YY->YY [0]; X-
-Y [eq]",

    "X[token]; Y[token]; XX[letter]; YY[letter]; X->XX; Y->YY; XX->XX [1]; YY->YY [1]" :
    "X[token]; Y[token]; XX[letter]; YY[letter]; X->XX; Y->YY; XX->XX [1]; YY->YY [1]; X-
-Y [eq]",

    "A1[token]; A2[token]; A3[token]; B()]; C1[token]; C2[token]; C3[token]; LINE[line];
LINE->A1->A2->A3->B->C1->C2->C3 [next_char]; C3->LINE [end_of_line];" :
    "LINE[line]; X[body]; Y[body]; Y->X [orbits]; X->A1 [p1]; X->A2 [p2]; X->A3[p3]; Y-
>C1 [p1]; Y->C2 [p2]; Y->C3 [p3]; A1[token]; A2[token]; A3[token]; C1[token]; C2[token];
C3[token];",

    "A[+1]; B[+10]; A->B [next_char]" :
    "A[+10]; B[+1]; A->B [next_char]",
    "A[+1]; B[+100]; A->B [next_char]" :
    "A[+100]; B[+1]; A->B [next_char]",
    "A[+1]; B[+1000]; A->B [next_char]" :
    "A[+1000]; B[+1]; A->B [next_char]",
    "A[+1]; B[+10000]; A->B [next_char]" :
    "A[+10000]; B[+1]; A->B [next_char]",
```

Here are some examples from the Advent of Code solutions. One technique I developed is to have left and right hand side of different lines so I could visually align them to highlight the changes. But think of like, 8 more rules exactly like this. You can see some rules get extremely hairy and a line break would really help.

MISTAKES?

Python is fine for me, but probably still the wrong language for embedding

- JavaScript?
- Lua?

I decided matches would be "injective" and then discovered cases where I didn't want that.

Can we make an implementation simple enough so that everybody can copy it?

- Tracery has ports in every language you can think of
- Probably not… I made heavy use of NetworkX and a constraint solver

Some things I'm less clear are mistakes, although they are signs that my language did not meet its design goal.
I like Python, but in terms of my design goals to create something embeddable, it's still too much. Lua would be a better choice, or maybe JavaScript.
Soffit ensures that each node on the left matches distinct nodes in the in the graph, because I thought that would be simpler to understand, but then I discovered cases where I wanted that! Like, if you're going to parse symbols and have only one version of each, suddenly injectivity means you can't match the pair "A, A".
Tracery has ports to every language you can think of; would it really be feasible to do this with Soffit? Not as it currently stands.

CATEGORY THEORY?

Algebra for graph rewriting

Double Pushout (DPO):

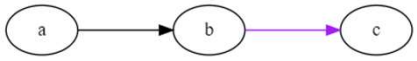
- Restricted deletion

Single Pushout (SPO):

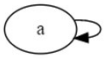
- Unrestricted deletion, "partial morphisms"

Other options: SQPO, AGREE, PBPO

No dangling edges



No half-dead nodes
(non-injective matchings only)



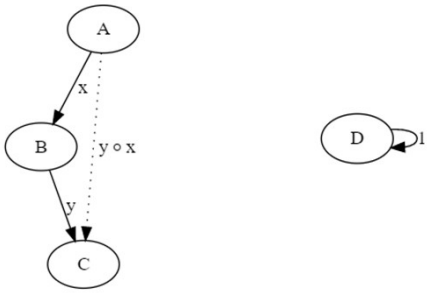
OK, let's talk about Category Theory, which provides a model for how graph grammars work. But really the practical impact can be expressed in this one page. A "DPO" is an older model that specifies how to handle deletion. You can't delete a vertex if it would leave an edge dangling, and you can't simultaneously delete a node or edge and not delete it. (This only occurs if nodes in the rule can match the same node in the graph.) Nearly everybody who writes about DPO will take pages to explain these two rules, but in practice they're pretty easy to understand.

A "SPO" operates on a more complicated category (that has partial functions instead of total functions) and allows unrestricted deletion.

A VERY SHORT INTRODUCTION TO CATEGORIES

A **category** is a collection of dots (objects) and arrows (morphisms), with the rules:

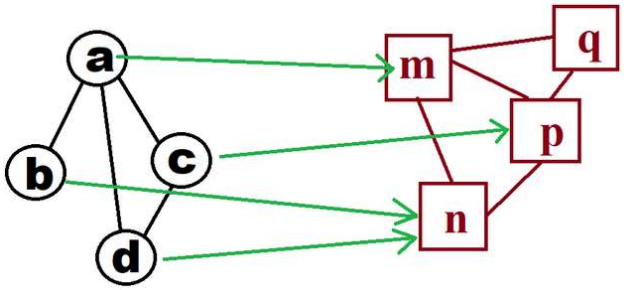
- Every pair of head-to-tail arrows can be joined to create a new arrow
- This rule is associative
- Every dot has an identity arrow



Every category is a multigraph, but not all graphs are categories.

THE CATEGORY OF GRAPHS

Here's one of its morphisms (the green arrows):

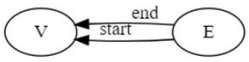


Note that this is not a function between arbitrary graphs, it's just a set of mappings between two *particular* graphs.

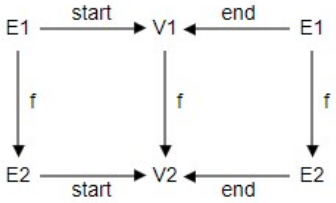
THE CATEGORY OF GRAPHS [2]

We can think of a directed graph as:

- a set of vertices
- a set of edges
- and a pair of functions "start" and "end" mapping edges to vertices.



A graph morphism maps vertices to vertices, and edges to edges, so that "start" and "end" in each graph match up.



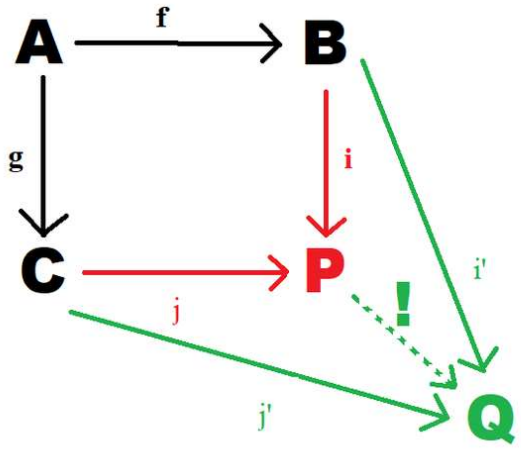
A **commutative diagram**: any two ways of combining the arrows are equal.

PUSHOUTS

A pushout is an object that has a "universal property".

Think of it as a machine:

- **Inputs:** A, B, C, and the morphisms f: A->B and g : A->C
- Outputs: P and the morphisms i:B->P and j:C->P that make a commutative square
- Special property: if ABCQ commutes too, then there's a unique morphism P->Q that makes everything commute.
- It's the "smallest" P that works.

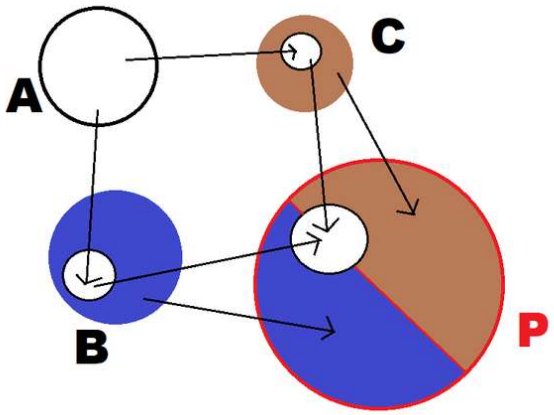


Pushouts aren't guaranteed in every category, but directed graphs have them.

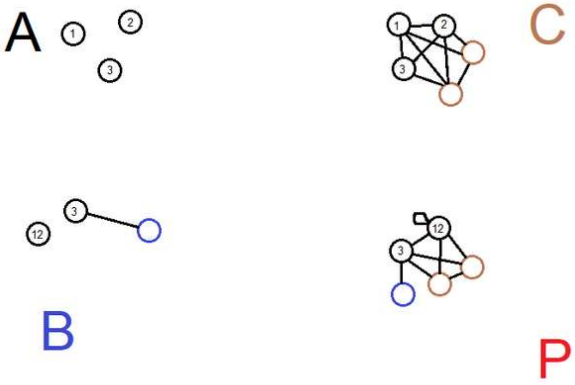
PUSHOUTS IN SET

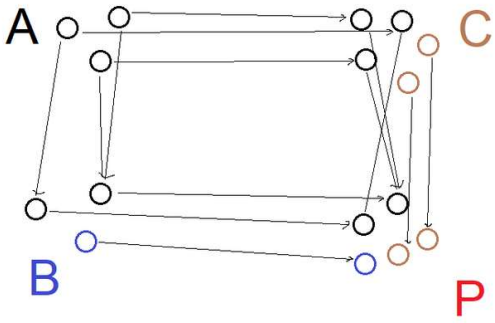
Start with a disjoint union

P "remembers" which of B or C each element came from, unless they were originally from A.



PUSHOUTS ON GRAPHS





P is the minimal graph that:

- contains (an image of) all of B
- contains all of C, but
- merges the image of A in both graphs

DOUBLE PUSHOUT

L is the left side of the rule

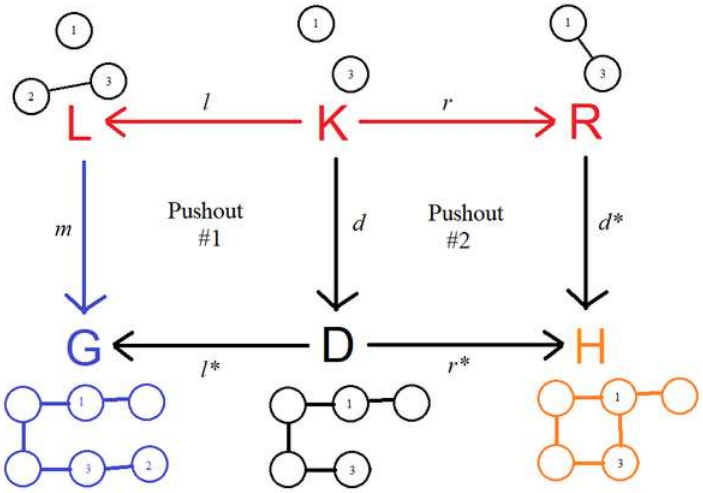
R is the right side of the rule

K is the left side but with elements deleted (so K->L is an injective map)

G is the input graph

D is the matched portion of G, with nodes deleted.

H is the result of performing all the addition in R.



Pushout #1 is the machine run in reverse. We want G to be the pushout, so we find a D that makes it one!
Pushout #2 is the single-pushout that doesn't have any deletion.

All this is just a fancy mathematical way of saying that the DPO rules hold.

THANKS!

Try Soffit online:

http://soffit.combinatorium.org or https://mgritter.github.io/soffit-web

- I've been meaning to link some examples, for now you can find them in my Gists: https://gist.github.com/mgritter

Check out the source code: `mgritter/soffit` on GitHub

These slides: `mgritter/minnebar-talks` on GitHub

Twitter: @markgritter

In RL:

WHAT'S NEXT FOR SOFFIT?

Format rewrite

- Keep the DOT-like syntax, but…
- Provide a way to name rules and add comments
- Compact representation for the "context"

Meta-rules

- Fuzzy idea: rules are graphs too, generate new rules via graph transformations?

```
"A->B [$n1]; C->D [$n2]": "...",
"X->Y [$n]" : ["X->Y[1]","X->Y[2]",...]
```