CLOJURE TOOLS FOR SYMBOLIC AI

Saul Johnson

Which tools specifically? - A Pattern Matcher

- A fully-featured symbolic pattern matcher for Clojure.
- In a nutshell: lets you flexibly iterate over collections of structured data and pull out information you’re interested in.
- An example problem: we’ve got a set of data about a bunch of different foods. **We need to know the name of every red vegetable.**

Red Vegetables: A Basic Example

We have data:

```
(def food
  '([isa cherry fruit]
    [isa cabbage veg]
    [isa chili veg]
    [isa radish veg]
    [isa leek veg]
    [color leek green]
    [color chili red]
    [color cherry red]
    [color cabbage green]
    [color radish red]))
```

Let’s match on it:

```
user=> (mfor ['[isa ?f veg]
food] (? f))
```

```
(cabbage chili radish leek)
```

```
user=> (mfor* ['([isa ?f veg]
[color ?f red]) food] (? f))
```

```
(chili radish)
```

Which tools specifically? – An Operator Search Mechanism

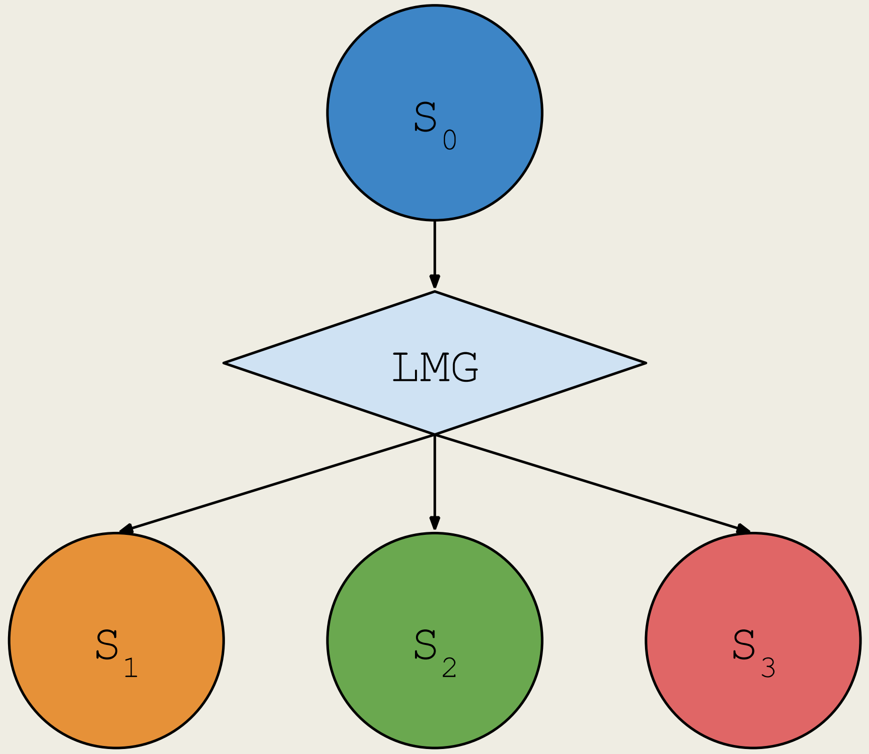
- A breadth-first search mechanism for applying operators.
- In a nutshell: lets you specify an initial state, a set of operators and a goal state, then plans how to realise your goal using your operators.
- An example problem: we need to move a keg of beer into the living room, but right now we're in the garage and the beer is in the kitchen under the celery. We can walk around, pick up and drop things. **What actions do we need to take?**

EXAMPLE TIME!

Let’s get the computer to do some planning for us!

How did that work? – Legal Move Generator (LMG)

A legal move generator (LMG) is a function which takes in our initial state and produces a number of successor states reachable in one move.



Let’s play countdown..!

- *Countdown* is a simple game we played in my maths class in upper school.
- You need to get from one number to another using only a limited number of mathematical operations. For example:
  - *Add 3*
  - *Multiply by 2*
  - *Subtract 7*
- Let’s capture this in an LMG.

And then cheat using an LMG!

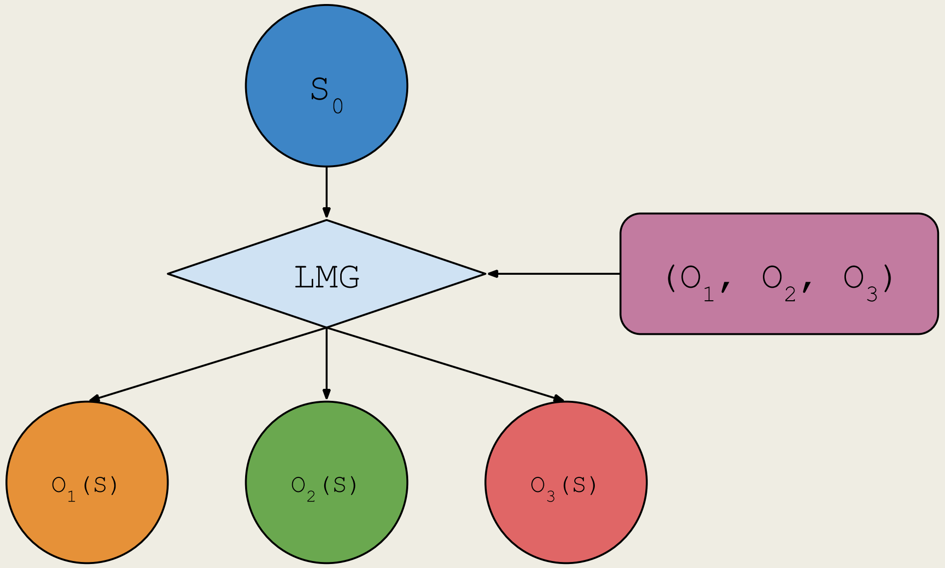
- In this example, our state is *n* (our current number). Our LMG applies each of our operators in turn to generate a collection of 3 successor states.
- We then run a breadth-first search on the resulting tree to plan a route to our goal.

```
(defn lmg [n]
 [(+ n 3) (* n 2) (- n 7)])

user=> (lmg 10)

[13 20 3]
```

```
user=> (breadth-search 5 35 lmg)

(5 8 16 32 35)
```

Getting More Complex with Operators

Instead of rolling our operators into our LMG, let's pass it a collection of STRIPS-style operators along with our initial state to generate our tree (or *state transition graph*).



STRIPS?

STRIPS (Stanford Research Institute Problem Solver) style operators specify:

- A set of preconditions: is this operator applicable in the current state?
- A set of additions to the state when the operator is applied.
- A set of deletions from the state when the operator is applied.

```
{:pre ((agent ?agent)
       (manipulable ?obj)
       (at ?agent ?place)
       (on ?obj ?place)
       (holds ?agent nil))
 :add ((holds ?agent ?obj))
 :del ((on ?obj ?place)
       (holds ?agent nil))}
```

Enter the Operator Search Mechanism!

Let’s use the operator search tool we mentioned to move the book from the table to the bench.

To the REPL!

```
#{(at R table)
  (on book table)
  (holds R nil)
  (path table bench)
  (manipulable book)
  (agent R)}
```

Expanding our Knowledge: Inference

Given a set of facts, what else can we infer?

If Pierre is Jack’s father and Jack is Mary’s father, we know that Pierre is Mary’s grandfather. We can infer this and add it to our fact base. This might allow us to infer something else, and so on.

Let’s capture that rule.

```
(def grandparent-rule
 '[rule 15
  [parent ?a ?b]
  [parent ?b ?c] =>
  [grandparent ?a ?c]])
```

Reusable Collection Transformations: Matcher Functions

We define a matcher function to compile our rule (which contains a set of antecedants and a set of consqeuents, separated by an arbitrary => notation) into map.

```
(defmatch compile-rule []
  ([rule ?id ??antecedents =>
    ??consequents] :=>
   {:id (? id)
    :ante (? antecedents)
    :consq (? consequents)}))
```

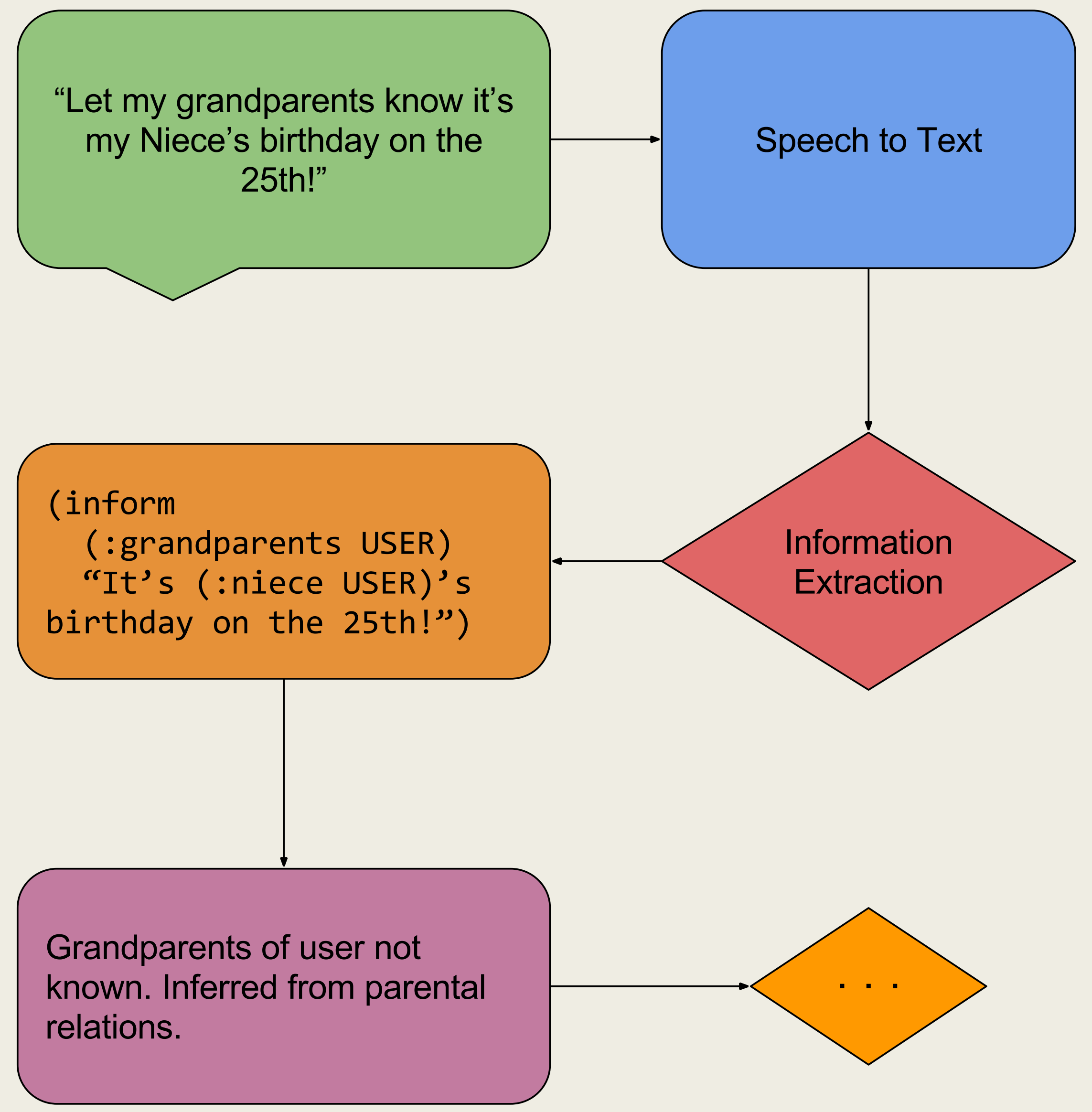
Time to Apply our Rule!

Let’s apply our grandparent rule to a set of parent relationships.

```
(def family
  '#{[parent Sarah Tom]
     [parent Steve Joe]
     [parent Sally Sam]
     [parent Ellen Sarah]
     [parent Emma Bill]
     [parent Rob Sally]})

(defn apply-rule [facts rule]
  (set (mfor* [(:ante rule) facts]
    (mout (:consq rule)))))

user=> (apply-rule family
  (compile-rule grandparent-rule))

#{[grandparent Ellen Tom]
  [grandparent Rob Sam]}
```

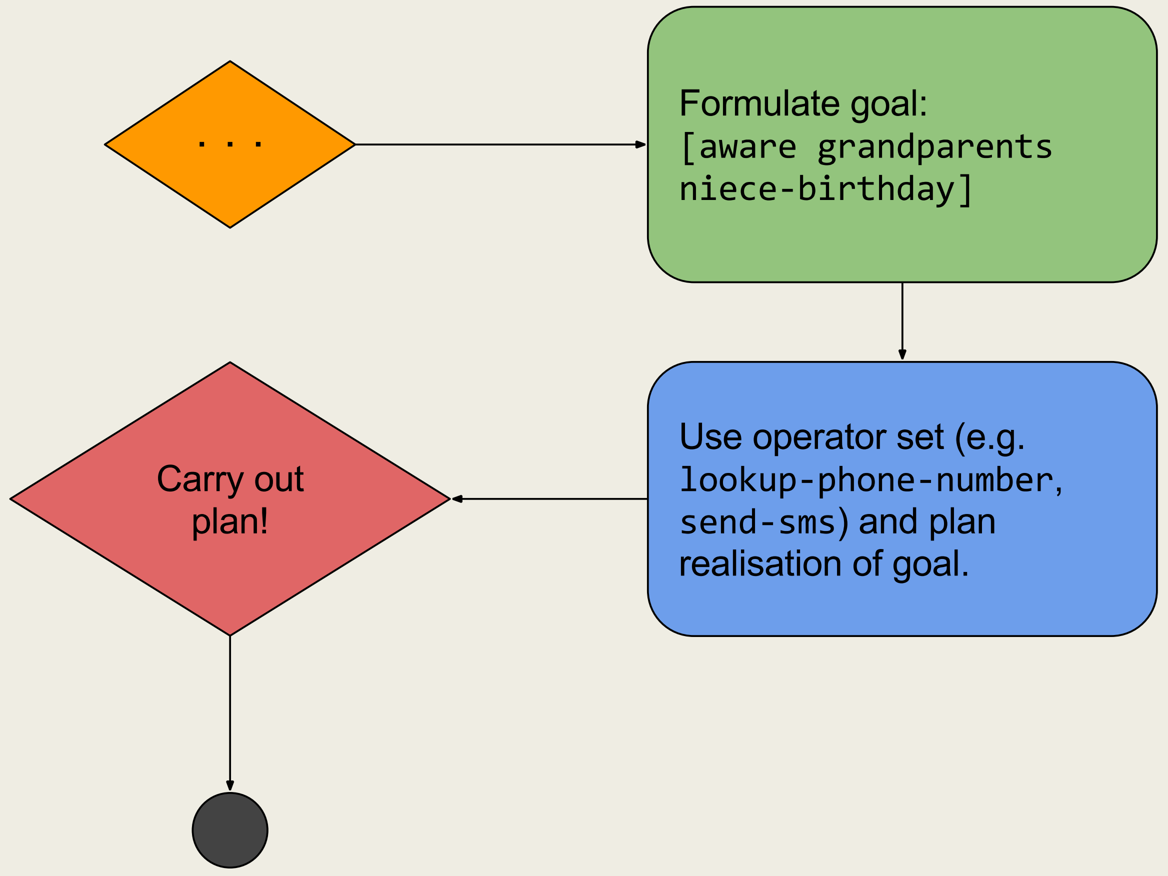
Bringing the Theory into Practice

How can we apply planning and inference engines to the real world? Let’s ask our phone’s voice assistant to do something…


“Let my grandparents know it’s my Niece’s birthday on the 25th!”
Speech to Text
Information Extraction
(inform (:grandparents USER) “It’s (:niece USER)’s birthday on the 25th!”)
Grandparents of user not known. Inferred from parental relations.
. . .

Bringing the Theory into Practice (cont.)

Using inference and planning we can begin to create quite a robust little AI!


. . .
Formulate goal:
[aware grandparents niece-birthday]
Use operator set (e.g. lookup-phone-number, send-sms) and plan realisation of goal.
Carry out plan!

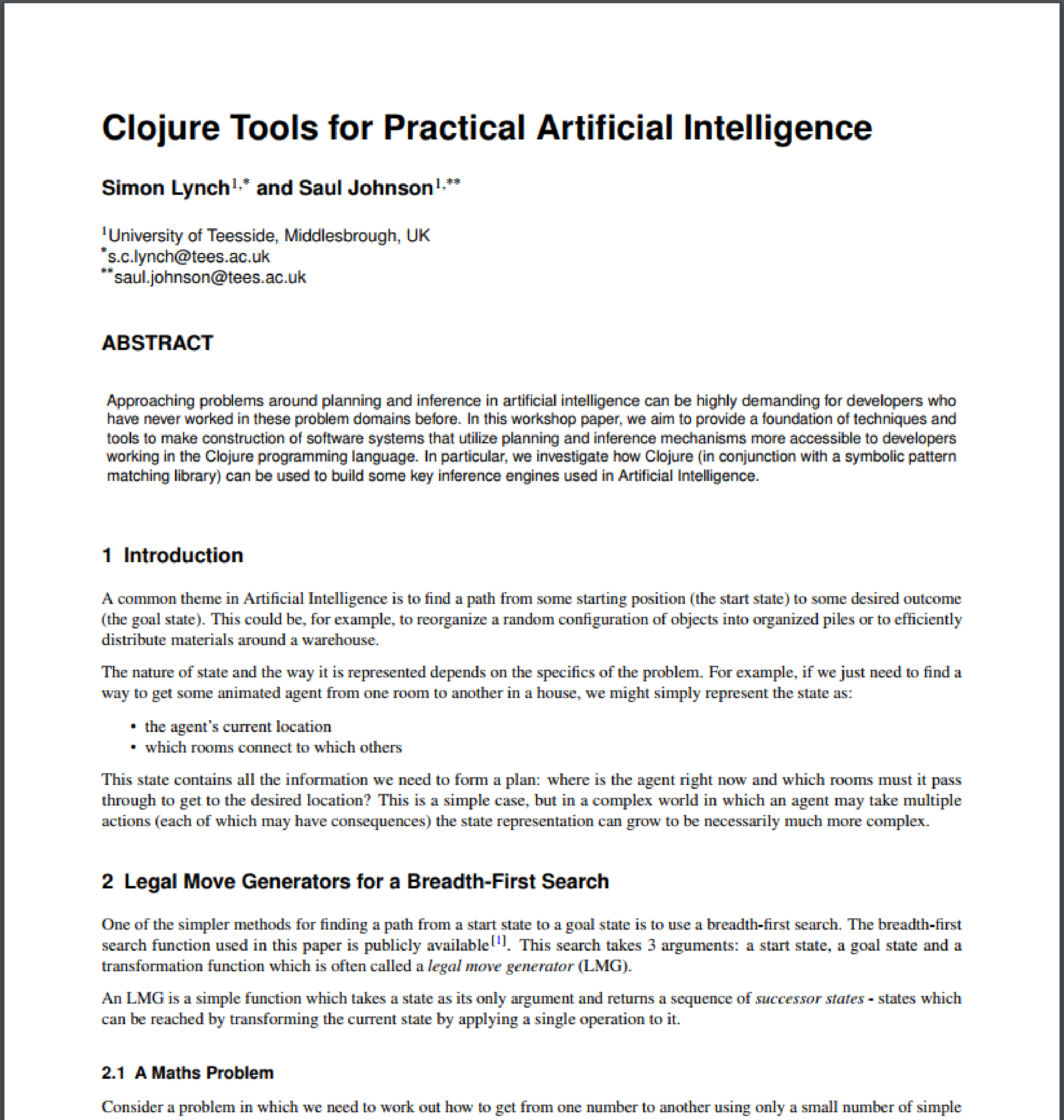
Get your hands on the code!

Releases are on Clojars, contribute/read the docs on GitHub, everything's on Leiningen!

This talk has a companion paper!

It goes into way more detail about the tools and techniques we've covered today:

https://cognesence.co.uk/euroclojure-2017



Clojure Tools for Practical Artificial Intelligence

Simon Lynch[1,*] and Saul Johnson[1,**]

[1]University of Teesside, Middlesbrough, UK
[*]s.c.lynch@tees.ac.uk
[**]saul.johnson@tees.ac.uk

ABSTRACT

Approaching problems around planning and inference in artificial intelligence can be highly demanding for developers who have never worked in these problem domains before. In this workshop paper, we aim to provide a foundation of techniques and tools to make construction of software systems that utilize planning and inference mechanisms more accessible to developers working in the Clojure programming language. In particular, we investigate how Clojure (in conjunction with a symbolic pattern matching library) can be used to build some key inference engines used in Artificial Intelligence.

1 Introduction

A common theme in Artificial Intelligence is to find a path from some starting position (the start state) to some desired outcome (the goal state). This could be, for example, to reorganize a random configuration of objects into organized piles or to efficiently distribute materials around a warehouse.

The nature of state and the way it is represented depends on the specifics of the problem. For example, if we just need to find a way to get some animated agent from one room to another in a house, we might simply represent the state as:

- the agent's current location
- which rooms connect to which others

This state contains all the information we need to form a plan: where is the agent right now and which rooms must it pass through to get to the desired location? This is a simple case, but in a complex world in which an agent may take multiple actions (each of which may have consequences) the state representation can grow to be necessarily much more complex.

2 Legal Move Generators for a Breadth-First Search

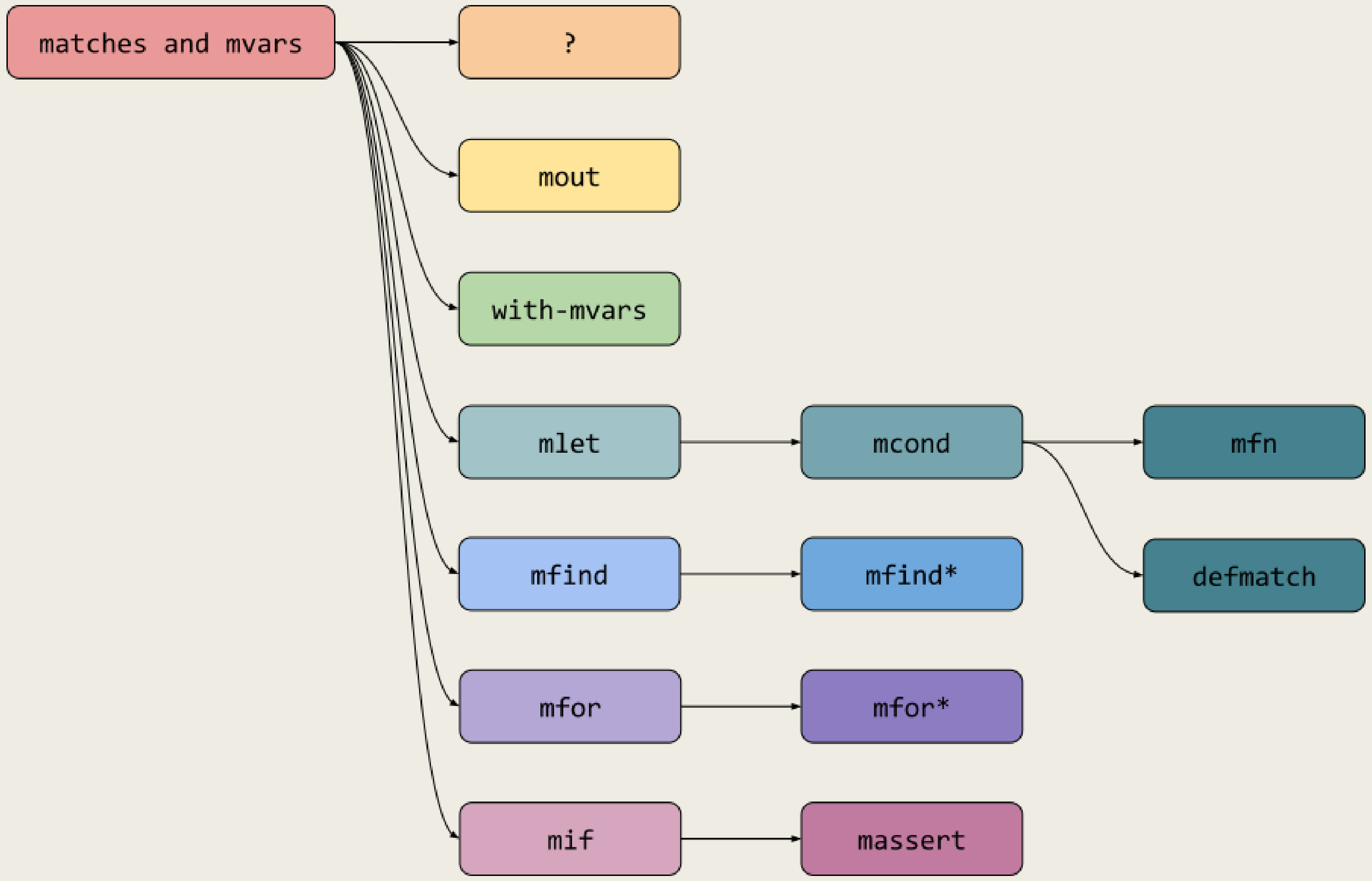
One of the simpler methods for finding a path from a start state to a goal state is to use a breadth-first search. The breadth-first search function used in this paper is publicly available[1]. This search takes 3 arguments: a start state, a goal state and a transformation function which is often called a *legal move generator* (LMG).

An LMG is a simple function which takes a state as its only argument and returns a sequence of *successor states* - states which can be reached by transforming the current state by applying a single operation to it.

2.1 A Maths Problem

Consider a problem in which we need to work out how to get from one number to another using only a small number of simple

Extras: The Matcher Form Family Tree


matches and mvars
?
mout
with-mvars
mlet
mcond
mfn
mfind
mfind*
defmatch
mfor
mfor*
mif
massert