CS61A Lecture 27
Logic Programming

Jom Magrotker
UC Berkeley EECS

August 2, 2012

Computer Science in the News

Adding a '3D print' button to animation software

July 31, 2012

Tool developed at Harvard turns animated characters into fully articulated action figures

CONTACT: Caroline Perry, (617) 496-1351

G.I. Joe may have finally met his match. (Photo courtesy of Moritz Bächer.)

http://www.seas.harvard.edu/news-events/press-releases/adding-a-3d-print-button-to-animation-software

TODAY

- Review: PyGic
- Unification

DECLARATIVE PROGRAMMING

New paradigm!

In ***declarative programming***, we describe ***what*** the properties of the required solution are, and the computer discovers how to find the solution.

Logic programming is a type of declarative programming that uses mathematical logic and logical inference to solve a problem.

http://www.5buckreview.com/wp-content/uploads/2011/03/Nimoy_Spock-284x300.jpg

REVIEW: PYGIC

Suppose we asserted the following fact:

```
P?> fact doctor(<christopher, david, matt>)
```

What would the interpreter print in response to the following queries?

```
P?> doctor(?who)

P?> doctor(<?who>)

P?> doctor(<?ninth, ?tenth, ?eleventh>)

P?> doctor(<?ninth | ?rest>)

P?> doctor(<christopher, ?tenth | ?eleventh>)
```

Review: PyGic

Suppose we asserted the following fact:

```
P?> fact doctor(<christopher, david, matt>)
```

What would the interpreter print in response to the following queries?

```
P?> doctor(?who)
?who = <christopher, david, matt>
P?> doctor(<?who>)
No.
P?> doctor(<?ninth, ?tenth, ?eleventh>)
?ninth = christopher, ?tenth = david, ?eleventh = matt
P?> doctor(<?ninth | ?rest>)
?ninth = christopher, ?rest = <david, matt>
P?> doctor(<christopher, ?tenth | ?eleventh>)
?tenth = david, ?eleventh = <matt>
```

PyGic: Rules

Conclusion

```
P?> rule grandfather(?person, ?grandson):
...     father(?person, ?son)
...     father(?son, ?grandson)
```

Hypotheses

The conclusion is true *only if* the hypotheses are true.

=

Can variables be replaced with values such that the hypotheses are true? If so, the conclusion is true too.

PyGic: Rules for Lists

We can also define rules for lists.

For example, say we want to check if two (flat) lists are equal.

What are some facts we know about equal lists?

PyGic: Rules for Lists

Fact 1:

The empty list is only equal to itself.

```
P?> fact equal_lists(<>, <>)
```

A fact, by the way, is equivalent to a rule with `True` in the body:

```
P?> rule equal_lists(<>, <>):
...      True
```

True is a PyGic keyword that is, well, true.

PyGic: Rules for Lists

Fact 2:

Conclusion

Two lists are equal if

their first elements are equal, and if

the rest of their elements are equal.

Hypotheses

```
P?> rule equal_lists(<?x | ?rest1>, <?x | ?rest2>):
...     equal_lists(?rest1, ?rest2)
```

The same variable is used in two places.
A list can therefore only match if the first elements have the same value.

PyGic: Rules for Lists

We want to append one list to another:

```
P?> append(<1, 2, 3>, <4, 5>, <1, 2, 3, 4, 5>)
Yes.
P?> append(<1, 2, 3>, <4, 5>, ?what)
Yes.
?what = <1, 2, 3, 4, 5>
```

PyGic: Rules for Lists

What are some facts we know about the problem?

Fact 1: Appending the empty list to any other list gives us the **other list**.

Fact 2: Appending one list to another is equivalent to adding the *first element of the first list* to the result of **appending the second list to the *rest* of the first list.**

What facts or rules should we then define?

PyGic: Rules for Lists

```
P?> fact append(<>, ?z, ?z)
```

We indicate that the first element of the first list and of the result *must* be the same.

```
P?> rule append(<?x | ?u>, ?v, <?x | ?w>):
...      append(?u, ?v, ?w)
```

PyGic: Rules for Lists

```
P?> fact append(<>, ?z, ?z)
```

What about the rest of the first list, the second list, and the result?

```
P?> rule append(<?x | ?u>, ?v, <?x | ?w>):
...      append(?u, ?v, ?w)
```

The result is the second list appended to the rest of the first list.

PyGic: Rules for Lists (Practice)

We can now run append "backwards".

```
P?> append(<1, 2, 3>, ?what, <1, 2, 3, 4, 5>)
Yes.
?what = <4, 5>

P?> append(?what, ?other, <1, 2, 3, 4, 5>)
Yes.
?what = <>
?other = <1, 2, 3, 4, 5>

P?> more?
Yes.
?what = <1>
?other = <2, 3, 4, 5>
```

PyGic: Rules for Lists (Practice)

Write the rule(s) `reverse` that will match only if the second list has elements in the reverse order as the first.

```
P?> reverse(<1, 2, 3>, <3, 2, 1>)
Yes.
P?> reverse(<1, 2, 3>, <1, 2>)
No.
P?> reverse(<1, 2, 3>, ?what)
Yes.
?what = <3, 2, 1>
```

(*Hint*: You may find append useful here.)

PyGic: Rules for Lists (Practice)

```
P?> fact reverse(<>, <>)

P?> rule reverse(<?first | ?rest>, ?rev):
...     reverse(?rest, ?rest_rev)
...     append(?rest_rev, <?first>, ?rev)
```

PyGic: Rules for Lists (Practice)

Write the rule(s) `palindrome` that will match only if the list is a palindrome, where the list reads the same forwards and backwards.

```
P?> palindrome(<1, 2, 3>)
No.
P?> palindrome(<1, 2, 3, 2, 1>)
Yes.
```

(*Hint*: You have defined `reverse` and `equal_lists`.)

PyGic: Rules for Lists (Practice)

```
P?> rule palindrome(?list):
...     reverse(?list, ?list_rev)
...     equal_lists(?list, ?list_rev)
```

ANNOUNCEMENTS

- Homework 13 due **Saturday, August 4**.
  - Includes Py, streams, iterators, and generators
  - Also includes the Project 4 contest.
- Project 4 due **Tuesday, August 7**.
  - Partnered project, in two parts.
  - Twelve questions, so *please start early*!
  - Two extra credit questions.
- De-stress potluck **tonight** from **7pm to 10pm** in the **Wozniak Lounge** (Soda, 4^{th} floor).
  - Food and games.
  - Come and leave when you want.

ANNOUNCEMENTS: FINAL

- Final is **Thursday, August 9**.
  - *Where*? 1 Pimentel.
  - *When*? 6PM to 9PM.
  - *How much*? *All* of the material in the course, from June 18 to August 8, will be tested.
- Closed book and closed electronic devices.
- One 8.5” x 11” ‘cheat sheet’ allowed.
- No group portion.
- We have emailed you if you have conflicts and have told us. If you haven’t told us yet, please *let us know* by toda
- Final review sessions on **Monday, August 6** and **Tuesday, August 7**, from **8pm to 9:30pm** in the HP Auditorium (306 Soda).

How PyGic Works

Assume that we have asserted these facts:

```
P?> fact father(james, harry)
Yes.
P?> fact father(harry, albus_severus)
Yes.
P?> fact father(harry, james_sirius)
Yes.
```

What happens in response to the query:

```
P?> father(?who, ?child)
```

How PyGic Works

PyGic starts off with a "global" empty frame.

There are no bindings in this frame yet.

HOW PYGIC WORKS

PyGic first searches for the facts that match the query and the rules whose conclusions match the query.

There are three such facts:

```
father(james, harry)
father(harry, albus_severus)
father(harry, james_sirius)
```

How PyGic Works

PyGic picks a fact:

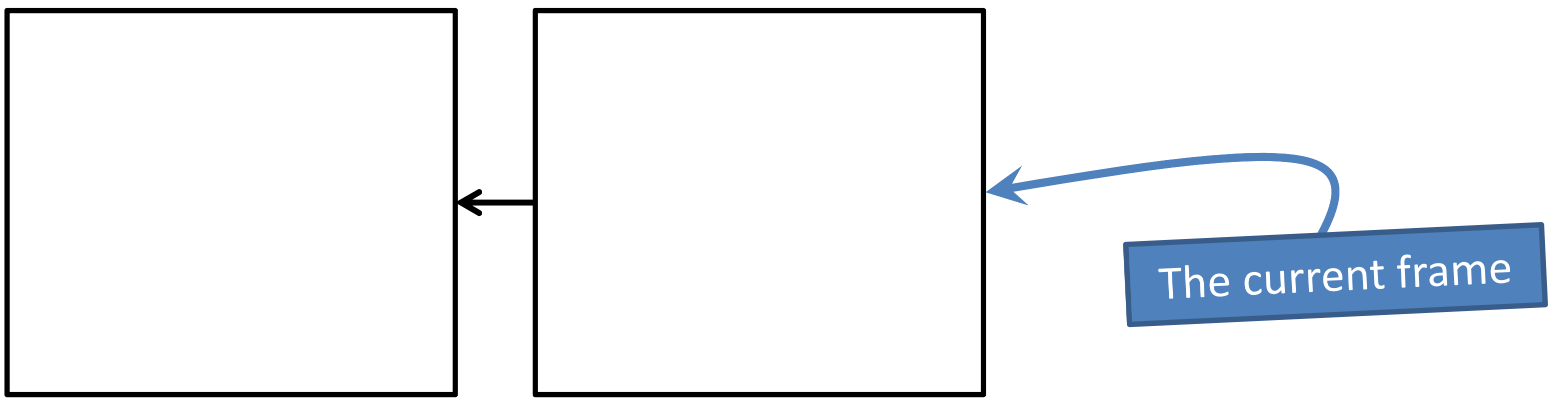
father(james, harry)

father(harry, albus_severus)

father(harry, james_sirius)

How PyGic Works

PyGic prepares an empty frame that extends the global frame.



It makes a new empty frame for every query.

How PyGic Works

PyGic attempts to ***unify*** the query with the fact.

Unification is a generalized form of pattern matching, where either or both of the patterns being matched may contain variables.

How PyGic Works: Unification

To match the query

`father(?who, ?child)`

with the fact

`father(james, harry)`,

PyGic must check if

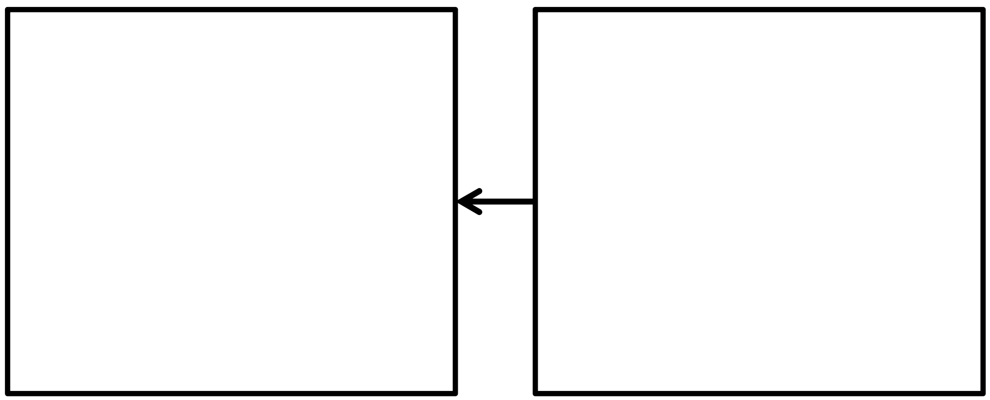
`?who = james, ?child = harry`.

How PyGic Works: Unification

PyGic checks if the variables

`?who` or `?child`

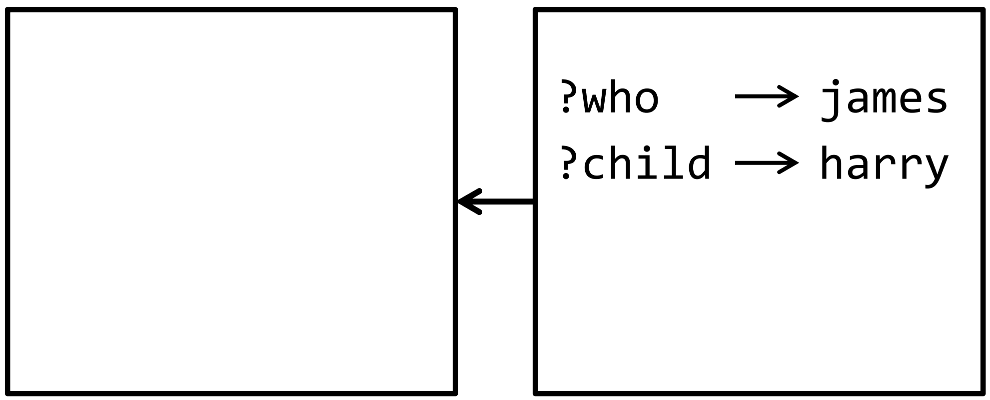
have any values in the current frame.



How PyGic Works: Unification

There are none!

So, PyGic binds the variables to these values in the current frame.



Now, trivially, we know that

`?who = james, ?child = harry`

is true.

How PyGic Works: Unification

PyGic is done with the query, since it successfully matched the query to a fact.

PyGic returns the frame and its bindings to be printed back to the user.

How PyGic Works: Backtracking

What happens if the user asks for `more`?

PyGic returns to the *last* point at which it made a choice (a ***choice point***), and ignores all the frames that it made as a result of that choice.

There, it tries to make another choice, *if it can*. If not, it goes to the choice point before that, and attempts to make another choice.

How PyGic Works: Backtracking

In this example, it made a choice when it chose which fact to unify. As a result, a new frame is created with possibly new bindings.

If it cannot choose another fact or rule to unify, and if there are no more choice points, then there are no more ways to satisfy the rules.

How PyGic Works: Backtracking

This is known as ***chronological backtracking***.

PyGic backtracks to the last point at which it made a choice and attempts to make another one to find another solution.

How PyGic Works

Now, say that we have the following rule:

```
P?> rule grandfather(?who, ?grandson):
...       father(?who, ?son)
...       father(?son, ?grandson)
```

What happens in response to the query:

```
P?> grandfather(james, ?grandson)
```

How PyGic Works

PyGic first searches for the facts that match the query and the rules whose conclusions match the query.

There is only one such rule:

`grandfather(?who, ?grandson)`

PyGic picks this rule.

HOW PYGIC WORKS

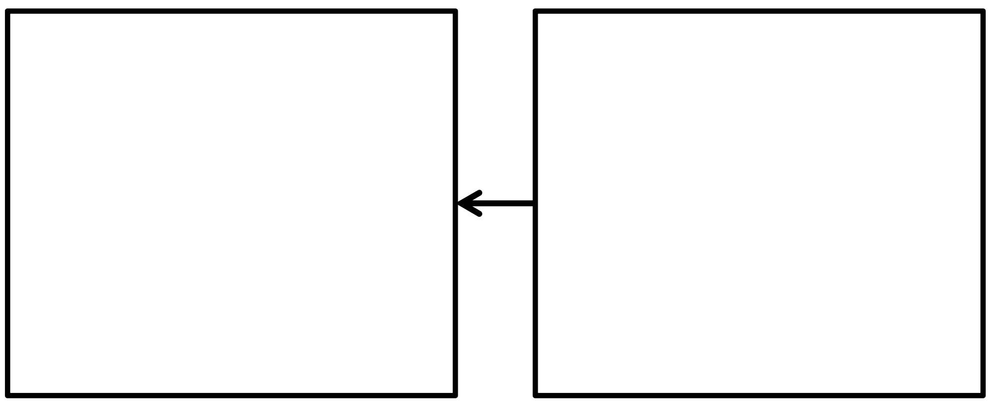
PyGic will rename the variables to avoid confusion with other rules that may have the same variable names.

The rule is now

```
grandfather(?who#1, ?grandson#2):
    father(?who#1, ?son#3)
    father(?son#3, ?grandson#2)
```

How PyGic Works

PyGic prepares an empty frame, where no variables have yet been bound to any value.



How PyGic Works: Unification

To match the query

`grandfather(james, ?grandson)`

with the rule conclusion

`grandfather(?who#1, ?grandson#2)`,

PyGic must check if

`?who#1 = james`,

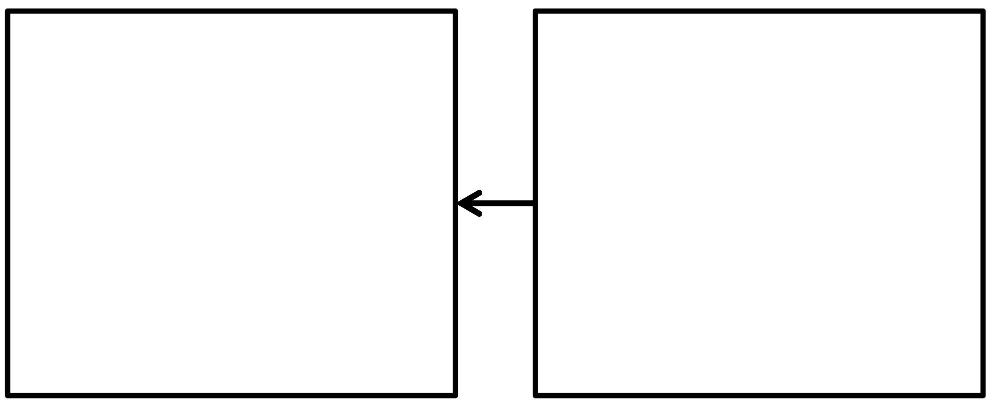
`?grandson#2 = ?grandson`.

How PyGic Works: Unification

PyGic checks if the variables

`?who#1, ?grandson#2, ?grandson`

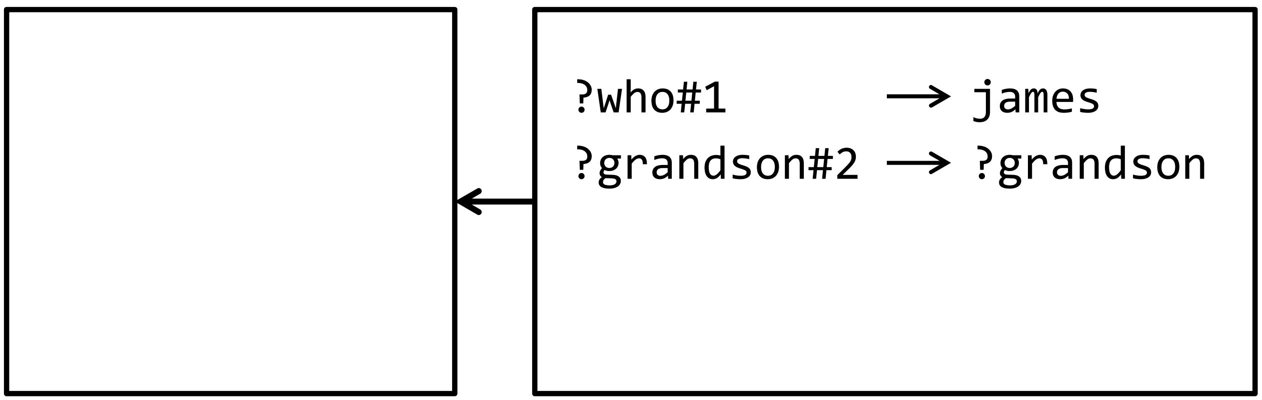
have any values in the current frame.



How PyGic Works: Unification

There are none!

So, PyGic makes the proper bindings in the current frame.



How PyGic Works: Hypotheses

However, this is a rule!

PyGic needs to determine if the hypotheses are true to infer that the conclusion is true.

PyGic will consider each hypothesis as a new query, and determine if each hypothesis is true.

How PyGic Works

The new query is

`father(?who#1, ?son#3)`

PyGic searches for facts and rule conclusions that match this query. There are three such facts:

```
father(james, harry)
father(harry, albus_severus)
father(harry, james_sirius)
```

How PyGic Works

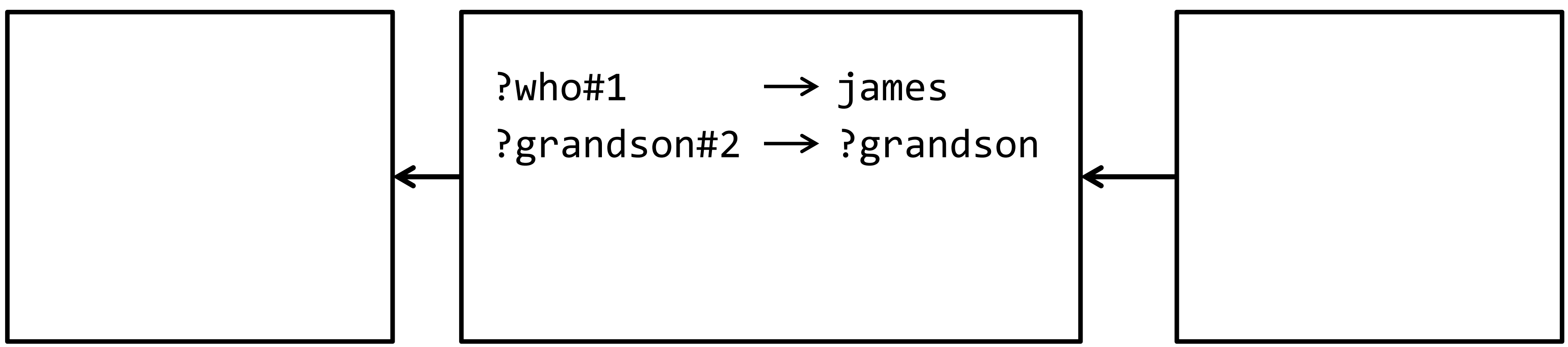
PyGic picks a fact:

father(james, harry)

```
father(harry, albus_severus)
father(harry, james_sirius)
```

How PyGic Works

PyGic prepares an *empty frame, which extends* the previous frame.



This is very different from the environment diagrams we studied earlier.
Variables could not be assigned to variables!

How PyGic Works: Unification

To match the query

`father(?who#1, ?son#3)`

with the fact

`father(james, harry)`,

PyGic must check if

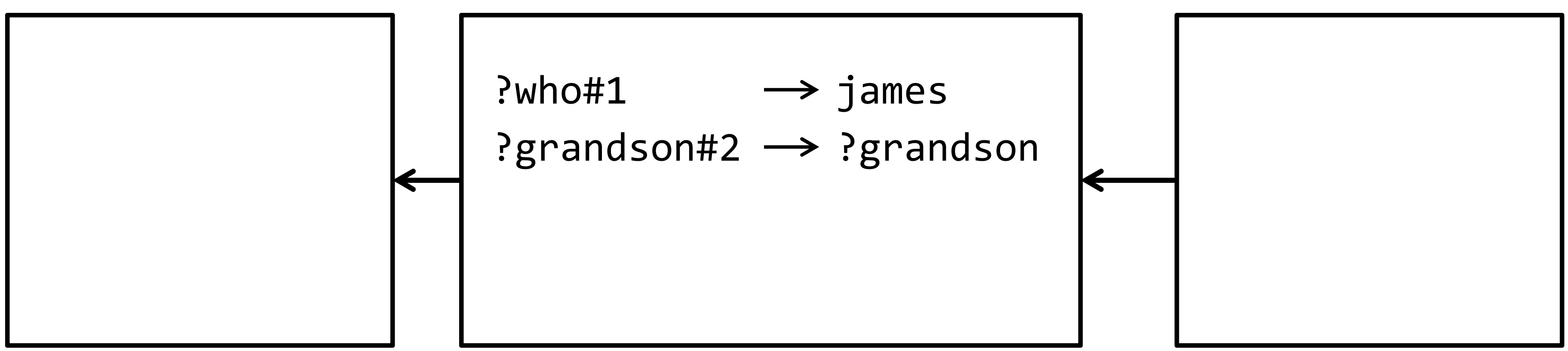
`?who#1 = james, ?son#3 = harry`.

How PyGic Works: Unification

PyGic checks if the variables

?who#1 or ?son#3

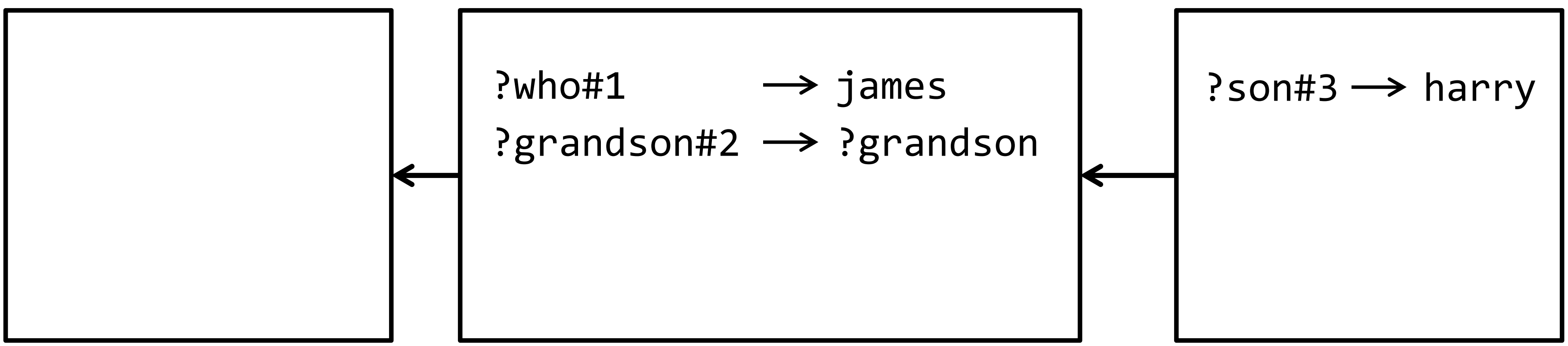
have any values in the current frame or its parent.



How PyGic Works: Unification

?who#1 has a value, but that value matches james.

There are no bindings for ?son#3.



PyGic binds ?son#3 with harry.

The query has been successfully unified with a fact.

How PyGic Works

The first hypothesis in the body of the `grandfather` rule is true. Now, we check the second hypothesis

```
father(?son#3, ?grandson#2)
```

PyGic searches for facts and rule conclusions that match this query. There are three such facts:

```
father(james, harry)
father(harry, albus_severus)
father(harry, james_sirius)
```

How PyGic Works

PyGic picks a fact:

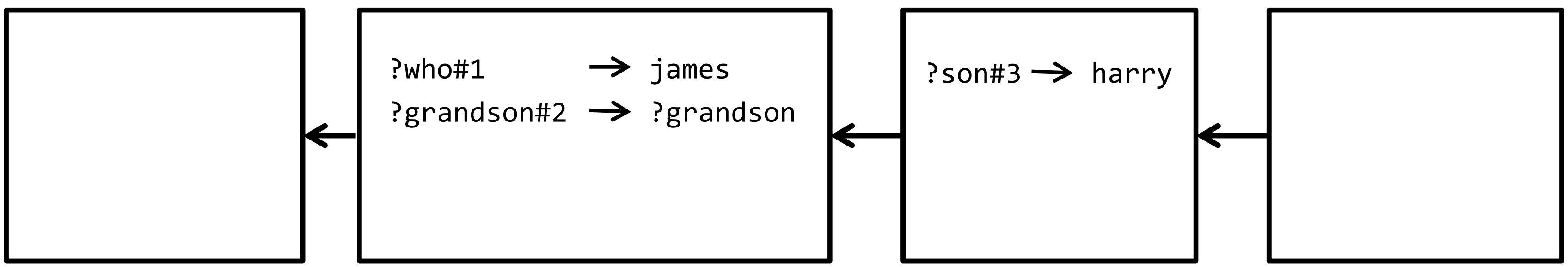
father(james, harry)

father(harry, albus_severus)

father(harry, james_sirius)

How PyGic Works

PyGic prepares another empty frame, *which extends* the previous frame.



How PyGic Works: Unification

To match the query

`father(?son#3, ?grandson#2)`

with the fact

`father(james, harry)`,

PyGic must check if

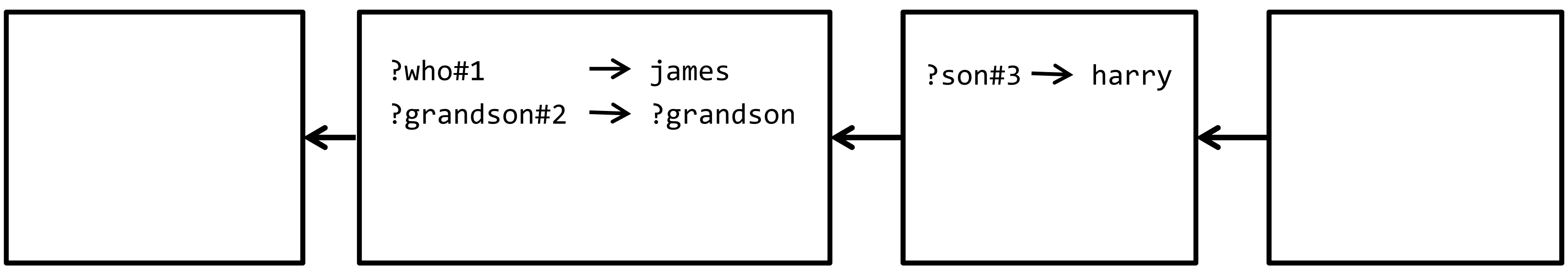
`?son#3 = james, ?grandson#2 = harry`.

How PyGic Works: Unification

PyGic checks if the variables

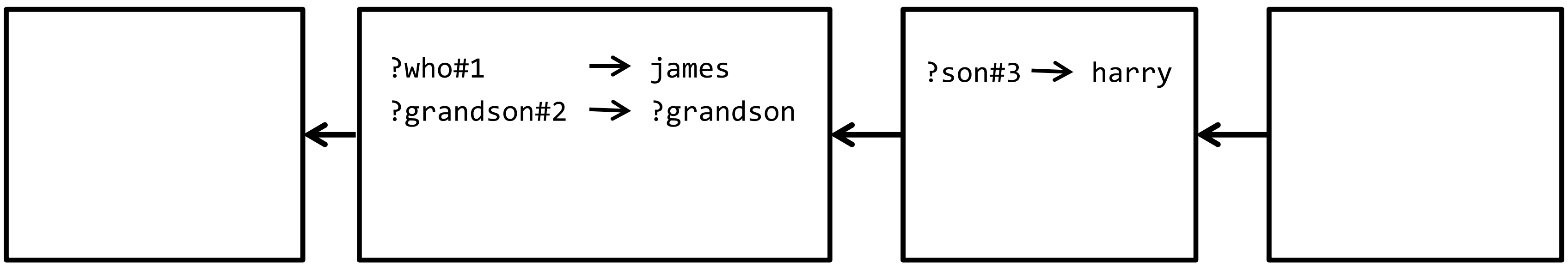
`?son#3` or `?grandson#2`

have any values in the current frame or its parents.



How PyGic Works: Unification

`?son#3` has a value, which does *not* match `james`.



The query is *not* true, given the existing bindings.

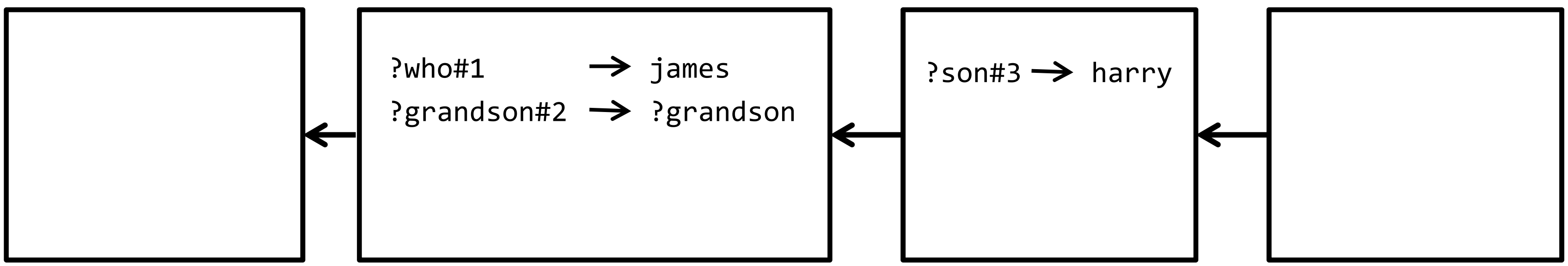
How PyGic Works

PyGic then backtracks and picks another fact:

```
father(james, harry)
father(harry, albus_severus)
father(harry, james_sirius)
```

How PyGic Works

PyGic prepares another empty frame, which extends the previous frame.



Notice that the "previous frame" is the frame *before* the last choice point.

How PyGic Works: Unification

To match the query

`father(?son#3, ?grandson#2)`

with the fact

`father(harry, albus_severus)`,

PyGic must check if

`?son#3 = harry`,

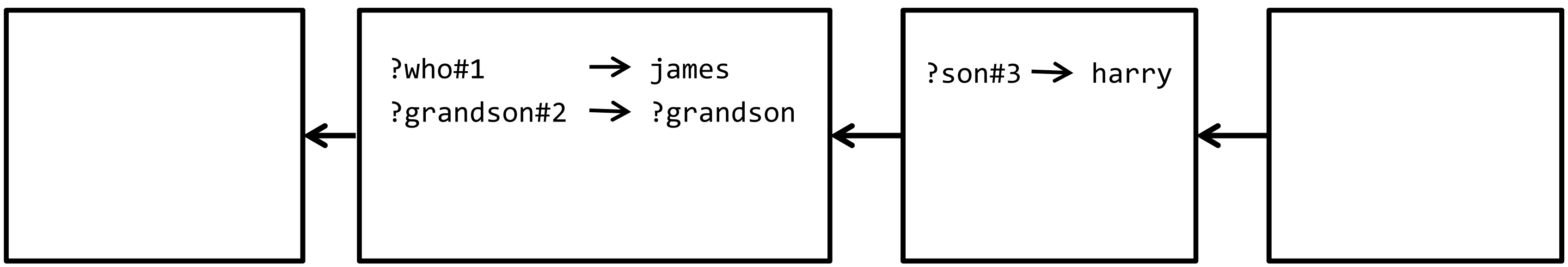
`?grandson#2 = albus_severus`.

How PyGic Works: Unification

PyGic checks if the variables

`?son#3` or `?grandson#2`

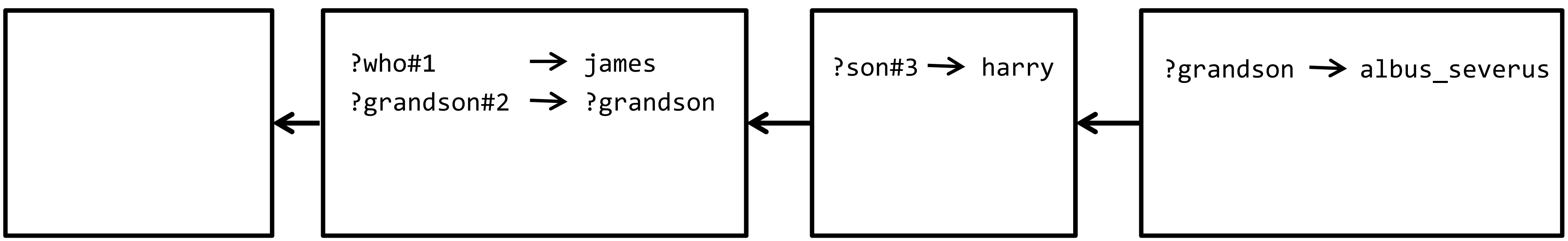
have any values in the current frame or its parents.



How PyGic Works: Unification

?son#3 has a value, but that value matches harry.

?grandson#2 has a value ?grandson.



PyGic binds ?grandson with albus_severus.

The query has been unified with the fact.

How PyGic Works

Both the hypotheses of the `grandson` rule are true, so the conclusion must also be true.

The values for the variables in the original query (`?grandson`) are looked up in the environment and printed.

How PyGic Works: Code (Simplified)

To prove a query in the given environment and with a given rule database...

```
def prove_query(query, env, ruledb):
    if is_true_expr(query):
        yield env
    else:
        matching_rules = ruledb.find_rules_matching(query)
        for rule in matching_rules:
            rule = rule.rename()
            newenv = pygic.environments.Environment(env)
            if unify_expr_lists(query, rule.conclusion, new_env):
                for result in prove_queries(rule.hypotheses, newenv, ruledb):
                    yield result
```

If the query is the true expression, yield the current environment.

Find all the rules in the database whose conclusions match the query. Remember that facts are *also* rules, each with a body of True.

How PyGic Works: Code (Simplified)

```
def prove_query(query, env, ruledb):
    if is_true_expr(query):
        yield env
    else:
        matching_rules = ruledb.find_rules_matching(query)
        for rule in matching_rules:
            rule = rule.rename()
            newenv = pygic.environments.Environment(env)
            if unify_expr_lists(query, rule.conclusion, new_env):
                for result in prove_queries(rule.hypotheses, newenv, ruledb):
                    yield result
```

For every rule that matches...

... returned a renamed rule.

How PyGic Works: Code (Simplified)

```
def prove_query(query, env, ruledb):
    if is_true_expr(query):
        yield env
    else:
        matching_rules = ruledb.find_rules_matching(query)
        for rule in matching_rules:
            rule = rule.rename()
            newenv = pygic.environments.Environment(env)
            if unify_expr_lists(query, rule.conclusion, new_env):
                for result in prove_queries(rule.hypotheses, newenv, ruledb):
                    yield result
```

Extend the environment.

Try to unify the query with the conclusion of the rule, which may add bindings in the frames.

If the hypotheses in the query can be proved, yield the environment that results from the proof.

How PyGic Works: Code (Simplified)

```
def prove_query(query, env, ruledb):
    if is_true_expr(query):
        yield env
    else:
        matching_rules = ruledb.find_rules_matching(query)
        for rule in matching_rules:
            rule = rule.rename()
            newenv = pygic.environments.Environment(env)
            if unify_expr_lists(query, rule.conclusion, new_env):
                for result in prove_queries(rule.hypotheses, newenv, ruledb):
                    yield result
```

`yield` allows us to continue where we left off, or to "backtrack" to a choice point.

How PyGic Works: Code (Simplified)

```
def prove_queries(queries, env, ruledb):
    if len(queries) == 0:
        yield env
    else:
        for new_env in prove_query(queries[0], env, ruledb):
            for result in prove_queries(queries[1:], new_env, ruledb):
                yield result
```

If there are no more queries to check, we successfully yield the current environment.

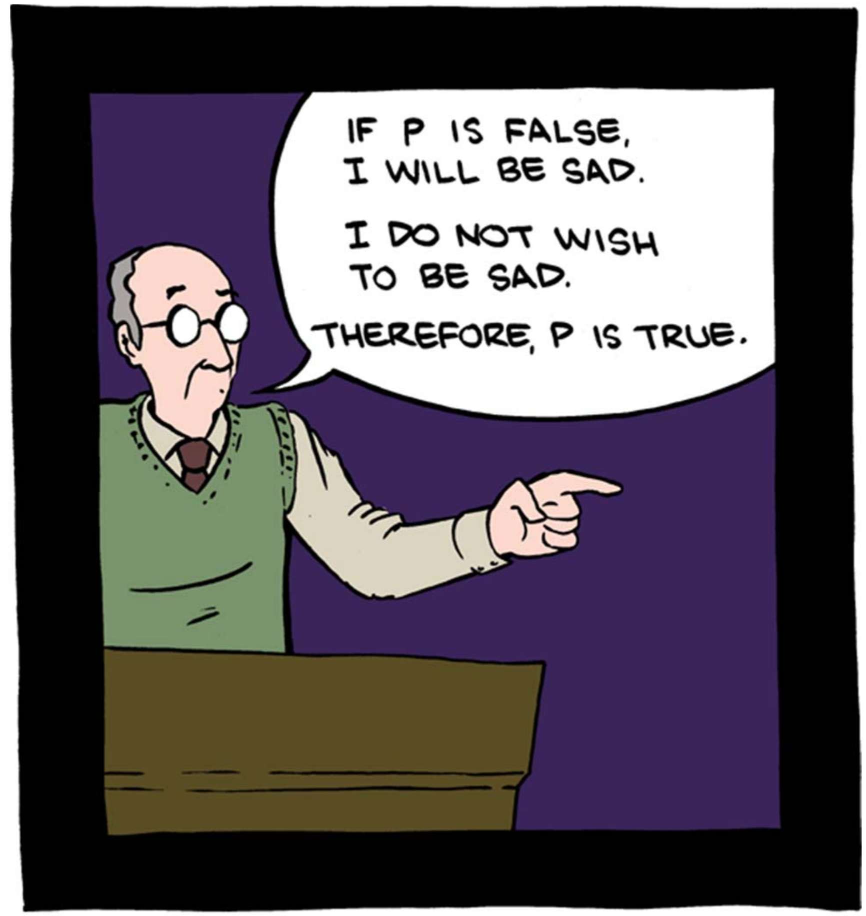
How PyGic Works: Code (Simplified)

```
def prove_queries(queries, env, ruledb):
    if len(queries) == 0:
        yield env
    else:
        for new_env in prove_query(queries[0], env, ruledb):
            for result in prove_queries(queries[1:], new_env, ruledb):
                yield result
```

Prove the first query and obtain a resulting new environment.

Prove the rest of the queries in the new environment.

BREAK



There. Now you can skip 99% of philosophical debates.

http://www.smbc-comics.com/index.php?db=comics&id=2673
http://www.smbc-comics.com/index.php?db=comics&id=2684

APPLICATIONS

Declarative programming is useful in database applications.

For example, if we have a database of student records, and want to get all the records of sophomore year students, we run the query

```
SELECT * FROM STUDENT_DB WHERE YEAR = 2
```

APPLICATIONS

Notice that in the query

```
SELECT * FROM STUDENT_DB WHERE YEAR = 2
```

all that we have specified are the properties we expect from our output.

We could, of course, iterate through all the records and filter out the ones we need, but it is a common enough operation that it is better to specify **what** we want from the output, rather than **how** we want to get it.

Conclusion

- Under the hood, PyGic matches the query against all of its rules and facts. It then picks one, and attempts to *unify* the query with the fact (or rule conclusion) by finding a consistent assignment to the variables in either.
- Declarative programming is useful in situations where we know what we expect of the output.
- ***Preview***: Write your own chat client.