Lecture 7

Implementing Prolog

unification, backtracking with coroutines

Ras Bodik

with Mangpo and Ali

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2013

UC Berkeley

Plan for today

More *programming under the abstraction*

developing abstractions that others can conveniently use

Previously, we extended a language with constructs

- iterators, lazy list concatenation, regexes
- mostly using coroutines

Today, we will build Prolog, an entirely new language

PA3 is assigned today: Prolog on top of your PA2 coroutines

In lecture today

Find a partner. Get a paper and pencil.

You will solve a series of exercises leading to a Prolog interpreter.

Prolog refresher

we can also define a food chain rule

chain(X,Y) :- eat(X,Y).
chain(X,Y) :- eat(X,Z), chain(Z,Y).

Program:

```
eat(ras, vegetables).
eat(ras, fruits).
eat(lion, ras).
```

Queries:

```
eat(ras, lion)?
eat(ras, X)?
```

eat(ras, lion)? → false

eat(ras, X)? → X = vegetables
X = fruits
no more answers

Structure of Programs

```
works(ras).                              Fact (Axiom)
works(thibaud) :- works(ras).            Rule
works(X)?                                Query
```

Variable

Constant
(atom)

Clause

In a rule:

Free Variable

```
a(X, Y) :- b(X,Z), c(Z,Y)
```

Head

Body

Variables in functional and logical programs

Functional programs

- values of expressions are bound to symbols (variables)
- environment: map from symbols to values
- symbols stay constant after created

Imperative programs

- as in functional, but binding can be changed later
- here, variables could be called “assignables”

Logic programs

- the role of symbol binding is replaced by *unification*
- note: unification will be undone during backtracking

Unification

Unification is what happens during matching

ie, during goal answering

unify(term1, term2) yields *most general unifier* (mgu)

```
a(1,Y)    |  a(X,2)
```

mgu: {Y ↦ 2, X ↦ 1}

```
a(X)      |  b(X)
```

false

```
a(1,Y)    |  a(2,X)
```

false = FAILURE

```
a(1, Y)   |  a(1, X)
```

mgu: {X ↦ Y}

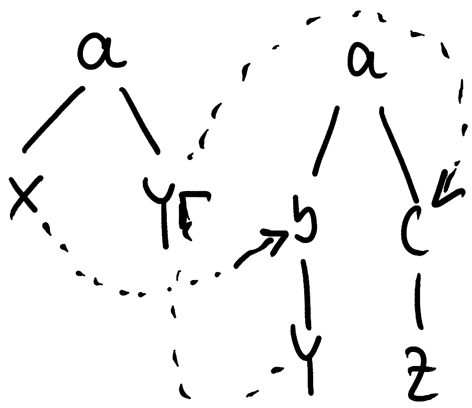
unify answers *false* when terms are not compatible

Exercise 1

Find the mgu for this unification:

a(X,Y) | a(b(Y),c(Z))

an mgu can be expressed in multiple ways



mgu:

$X \mapsto b(c(Z))$

$Y \mapsto c(Z)$

mgu:

$X \mapsto b(Y)$

$Y \mapsto c(Z)$

Lists

Lists are written [a, b, c]

which is the same as [a | [b,c]]

using the notation [Head | Tail]

so [a, b, c] is really desugared fully to [a | [b | [c | []]]]

The notation [H | T] is itself sugar for .(H,T)

Exercise 2

Note: this infinitely large mgu is not in Prolog, which needs finite mgus.

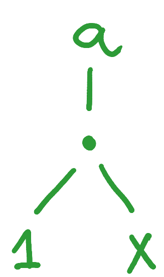
Find mgu for this unification:

t_1 $\qquad$ t_2

`a([1|X]) | a(X)`

desugars to

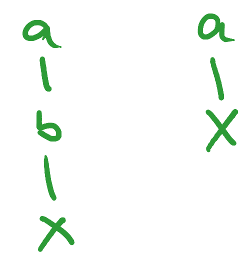
t_1:



t_2:

if lists are confusing here is a simpler example showing the same

$a(b(X)) \mid a(X)$



$X = b(b(b(\ldots)))$

we are asking "X should be equal to what tree so that terms t_1 and t_2 unify?"

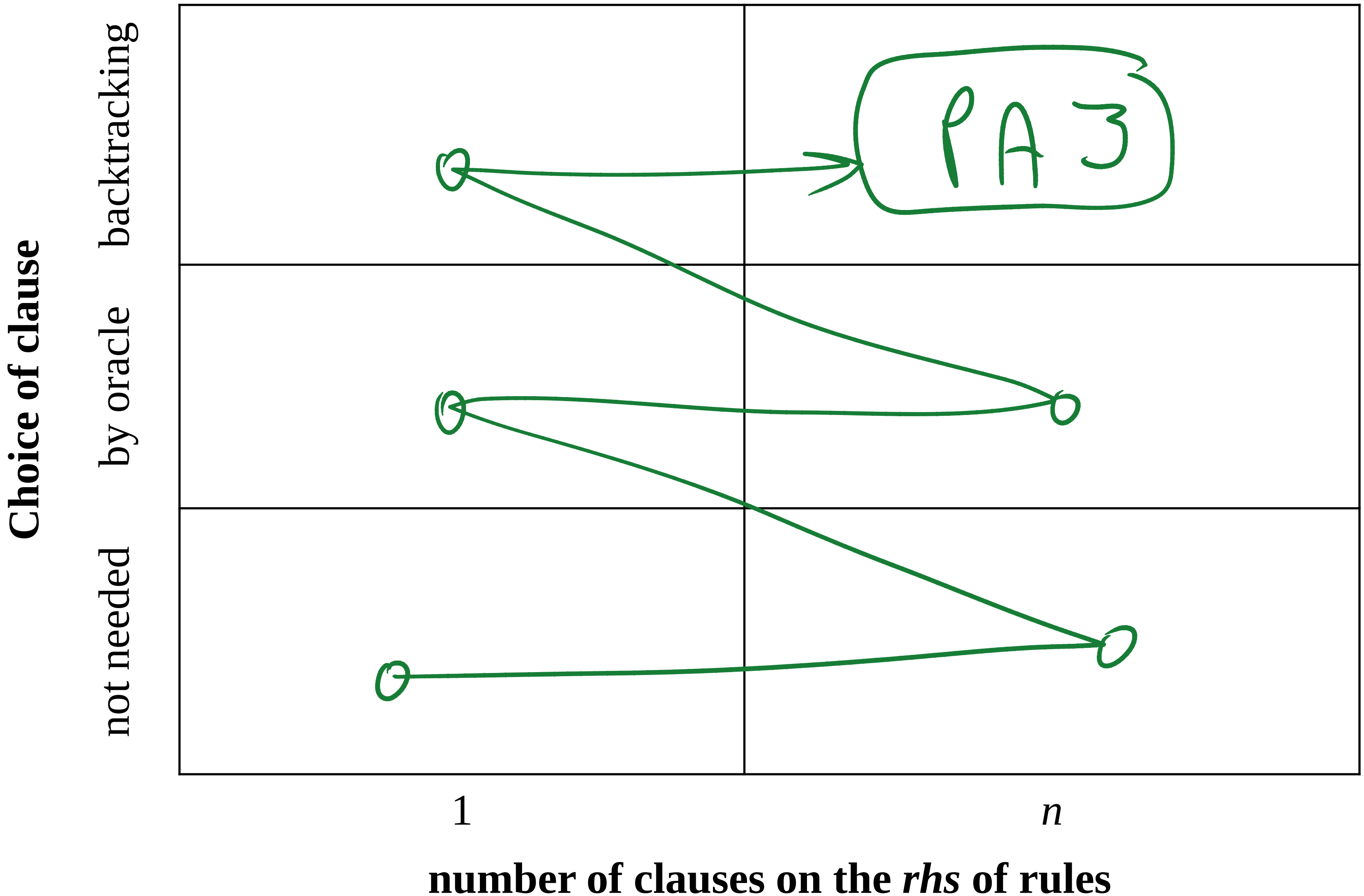
Answer: $[1,1,\ldots]$ infinite stream of 1.

The unification algorithm

See the simple description in *The Art of Prolog*

- Chapter 4.1, pages 88-91.
- on reserve in the library

Today, you will design a series of algorithms



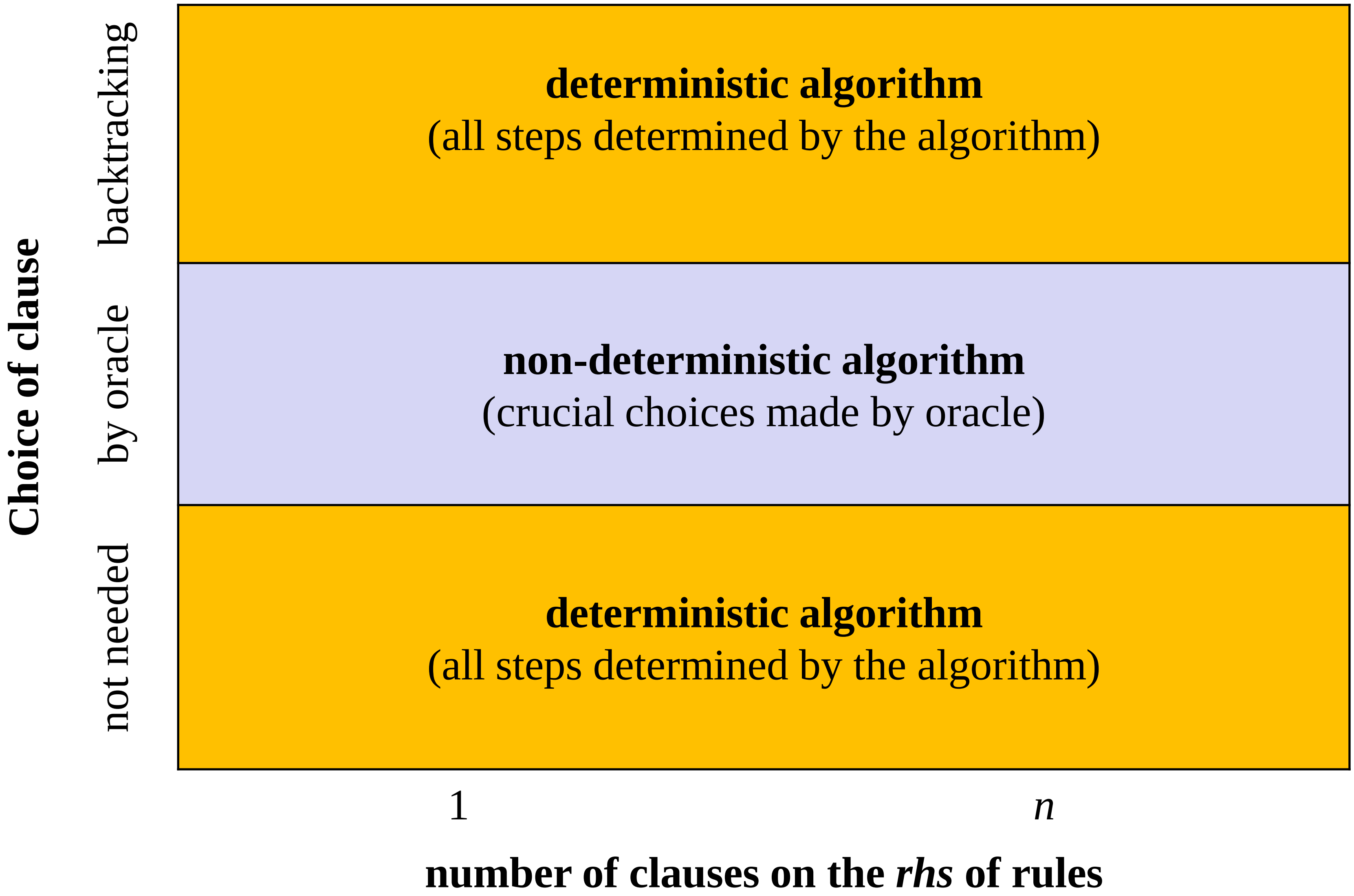
We will start with subsets of Prolog

Choice of clause	1	*n*
backtracking by oracle	a(X) :- b(X). b(Y) :- c(Y). c(1). c(2).	a(X) :- b(X), c(X). b(2). c(1). c(2).
not needed	a(X) :- b(X). b(Y) :- c(Y). c(1).	a(X) :- b(X), c(X). b(1). c(1).

number of clauses on the *rhs* of rules

more general

Some algorithms will use “magic”



Algorithm (1, no choice)

Choice of clause	1	*n*
backtracking		
by oracle		
not needed	a(X) :- b(X). b(Y) :- c(Y). c(1).	

number of clauses on the *rhs* of rules

Prolog execution is finding a proof of query truth

Program:

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).
```

Goal (query):

```
?- a(Z).
```

Answer:

```
true
Z = 1
```

Proof that the query holds:

c(1)	base fact, implies that …
c(Y)	holds, which implies that …
b(Y)	holds, which implies that …
b(X)	holds, which implies that …
a(X)	holds, which implies that …
a(Z)	holds.

The last one is the query

so the answer is true!

Recall "c(Y) holds" means

exists value for Y such that C(Y) holds.

Proof tree

Program:

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).
```

Goal (query):

```
?- a(Z).
```

Answer:

```
true
Z = 1
```

These steps form a proof tree

```
a(Z)
a(X)
  b(X)
  b(Y)
    c(Y)
    c(1)
      true
```

N.B. this would be a proof tree, rather than a chain, if rhs's had multiple goals.

Let's trace the process of the computation

Program:

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).
```

Goal (query):

```
?- a(Z).
```

Answer:

```
true
Z = 1
```

Two operations do all the work:

`a(Z)`	the query is our initial goal
`a(X)`	match head of a(X):-b(X)
`b(X)`	reduce goal a(X) to goal b(X)
`b(Y)`	match head of b(Y):-c(Y)
`c(Y)`	reduce b(Y) to c(Y)
`c(1)`	match head of c(1).
true	*we matched a fact*

The operations:

1) match goal to a head of clause C
2) reduce goal to rhs of C

matches produce mgus

rhs rewrite goals to new subgoals

Now develop an outline of the interpreter

Student answer:

procsss (A : goal)

for each head H of a clause C

if unify (A, H):

new_goal = H

new_goal = C.rhs

return process (new_goal)

if ¬RHS return OK.

return fail

Algorithm (1,no choice) w/out handling of mgus

```
def solve(goal):
    match goal against the head C.H of a clause C
    // how many matches are there? Can assume 0/1
    if no matching head found:
        return FAILURE  // done
    if C has no rhs:
        return SUCCESS  // done, found a fact
    else    // reduce the goal to the rhs of C
        return solve(C.rhs)
```

Note: we ignore the handling of mgus here, to focus on how the control flows in the algorithm. We'll do mgus next …

Concepts: Reduction of a goal. Unifier.

We reduce a goal to a subgoal

- If the current goal matches the head of a clause C, then we reduce the goal to the rhs of C.

Result of solving a subgoal is a unifier (mgu)

- or false, in the case when the goal is not true

But what do we do with the unifiers?

- are these mgus merged? If yes, when?

An algorithmic question: when to merge mgus

Program:

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).
```

Goal (query):

```
?- a(Z).
```

Answer:

```
true
Z = 1
```

Unifications created in matching

```
a(Z)
a(X)        Z=X
  b(X)
  b(Y)      X=Y
    c(Y)
    c(1)    Y=1
      true
```

Result is conjunction of these mgus:

Z=X, X=Y, Y=1

So, the answer is Z=1

variables X,Y are suppressed in answer

Design question: How do MGUs propagate?

Down the recursion? or …

```
a(Z)
a(X)          Z=X
  b(X)
  b(Y)        Z=X, X=Y
    c(Y)
    c(1)      Z=X, X=Y, Y=1
      true
```

merge mgus
on the way down?

MGUs propagate the answer

… up the recursion or …

a(Z)

a(X) $\boxed{Z=X}$ Z=X,X=Y,Y=1

b(X)

b(Y) $\boxed{X=Y}$ X=Y,Y=1

c(Y)

c(1) $\boxed{Y=1}$ Y=1

true

merge mgus on the way up?

MGUs propagate the answer

… or both?

`a(Z)`		
`a(X)`	`Z=X`	`Z=X,X=Y,Y=1`
`  b(X)`		
`  b(Y)`	`Z=X,X=Y`	`Z=X,X=Y,Y=1`
`    c(Y)`		
`    c(1)`	`Z=X,X=Y,Y=1`	`Z=X,X=Y,Y=1`
`      `*`true`*		

accumulate mgus

propagate constraints up

Both up and down propagation is needed

Consider program:

```
a(X,Y,Z) :- b(X,Y,Z).
b(A,B,C) :- c(A,B), d(C).
c(1,2).
d(1).
```

Down propagation: needed to propagate constraints

given query `a(X,X,Z)`?, goal c(X,Y) must be reduced to c(X,X) so that match with c(1,2) fails

Up propagation: needed to compute the answer to q.

given query `a(X,Y,Z)`?, we must show that Z=1 is in the result. So we must propagate the mgus up the recursion.

Algorithm (1,no choice) with unification, style 1

```
solve(goal, mgu):
    // match goal against the head C.H of a
    // clause C,  producing a new mgu.
    // unify goal and head wrt constraints in mgu
    mgu = unify(goal, head, mgu)
    if no matching head found:    old
new     return nil  // nil signifies FAILURE
    if C has no rhs:
        return mgu  // this signifies SUCCESS
    else
        // solve and return the updated mgu
        return solve(C.rhs, mgu)
```

Algorithm (1,no choice) with unification, style 2

```
solve(goal):
   // mgus've been substituted into goal and head
   mgu = unify(goal,head)
   if no matching head found:
        return nil  // nil signifies FAILURE
   if C has no rhs:
        return mgu  // this signifies SUCCESS
   else
        sub_goal = substitute(mgu,C.rhs)
        sub_mgu = solve(sub_goal)
        return merge(mgu, sub_mgu)
```

Unify and subst used in PA3

unify: Are two terms compatible? If yes, give a <u>unifier</u>

a(X, Y) | a(1, 2) --> {X -> 1, Y -> 2}

subst: Apply Substitution on clauses

subst[a(X, Y), {X -> ras, Y -> Z}] --> *a(ras, Z)*

$X \mapsto Y$

$Y \mapsto Z$

$Z \mapsto 1$

u

t $a(X, Y, Z)$

subst u into t

$a(1, 1, 1)$

Summary of Algorithm for (1, no choice)

The algorithm is a simple recursion that reduces the goal until we answer true or fail.

- the match of a goal with a head produces the mgu

The answer is the most general unifier

- if the answer is true
- mgus are unified as we return from recursion

This algorithm is implemented in the PA3 starter kit

Discussion

Style 1:

unify() performs the substitution of vars in goal, head based on the mgu argument. This is expensive.

Style 2:

mgus are substituted into new goals. This is done just once.

But we need to merge the mgus returned from goals.

This merge always succeeds (conflicts such as X=1, X=2 can't arise)

PA3 uses the second style.

In the rest of the lecture, we will abstract mgus.

You'll retrofit handling of mgus into algorithms we'll cover.

Example executed on PA3 Prolog

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).

a(I)?
```

```
Goal: a(I)
Unify: a(X_1) and a(I)
   Unifier: {X_1->I  }
Goal: b(I)
Unify: a(X_2) and b(I)
   Unifier: null
Unify: b(Y_3) and b(I)
   Unifier: {Y_3->I  }
Goal: c(I)
Unify: a(X_4) and c(I)
   Unifier: null
Unify: b(Y_5) and c(I)
   Unifier: null
Unify: c(1) and c(I)
   Unifier: {I->1  }
I = 1
```

```
Asking for solution 2
Unify: c(1) and b(I)
   Unifier: null
Unify: b(Y_8) and a(I)
   Unifier: null
Unify: c(1) and a(I)
   Unifier: null
None
```

Algorithm (n, no choice)

Choice of clause	1	*n*
backtracking		
by oracle		
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	a(X) :- b(X), c(X). b(1). c(1).

number of clauses on the *rhs* of rules

Resolvent

Resolvent: the set of goals that need to be answered

with one goal on rhs, we have always just one pending goal

Resolvent goals form a stack. The algorithm:

1) pop a goal
2) finds a matching clause for a goal, as in (1, no choice)
3) if popped goal answered, goto 1
4) else, push goals from rhs to the stack, goto 1

This is a conceptual stack.

Need not be implemented as an explicit stack

Algorithm

For your reference, here is algorithm (1,no choice)

```
solve(goal):
      match goal against the head C.H of a clause C

      if no matching head found:
            return FAILURE
      if C has no rhs:      // C is a fact
            return SUCCESS
      else        // reduce the goal to the rhs of C
            return solve(C.rhs)
```

Student algorithm

What to change in (n, no choice)?

```
solve(goal):
    match goal against a head C.H of a clause C

    if no matching head found:
        return FAILURE
    if C has no rhs:  // C is a fact
        return SUCCESS
    else    // reduce goal to the goals in the rhs of C
        for each goal in C.rhs
            if solve(goal) == FAILURE
                return FAILURE
        end for
        // goals on the rhs were solved successfully
        return SUCCESS
```

Your exercise

Add handling of mgus to (n, no choice)

Summary

The for-loop across rhs goals effectively pops the goals from the top of the conceptual resolver stack

This stack is comprised of all rhs rules to be visited by the for loops on the call stack of the algorithm.

Example executed on PA3 Prolog

```
a(X) :- b(X), c(X).
b(1).
c(1).

a(I)?
```

Asking for solution 1
Goal: a(I)
Unify: a(X_1) and a(I)
  Unifier: {X_1->I }
Goal: b(I)
Unify: a(X_2) and b(I)
  Unifier: null
Unify: b(1) and b(I)
  Unifier: {I->1 }
Goal: c(1)
Unify: a(X_4) and c(1)
  Unifier: null
Unify: b(1) and c(1)
  Unifier: null
Unify: c(1) and c(1)
  Unifier: {}
I = 1

Asking for solution 2
Unify: c(1) and b(I)
  Unifier: null
Unify: b(1) and a(I)
  Unifier: null
Unify: c(1) and a(I)
  Unifier: null
None

Algorithm (1, oracular choice)

Choice of clause	1	*n*
backtracking		
by oracle	a(X) :- b(X). b(Y) :- c(Y). c(1). c(2). goal c(Y) can match multiple heads	
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	Concept: **resolvent** Implementation: **recursion deals with reduced goals; iteration deals with rhs goals**

number of clauses on the *rhs* of rules

Search tree

First, assume we want just one solution (if one exists)

- ie, no need to enumerate all solutions in this algorithm

We'll visualize the space of choices with a search tree

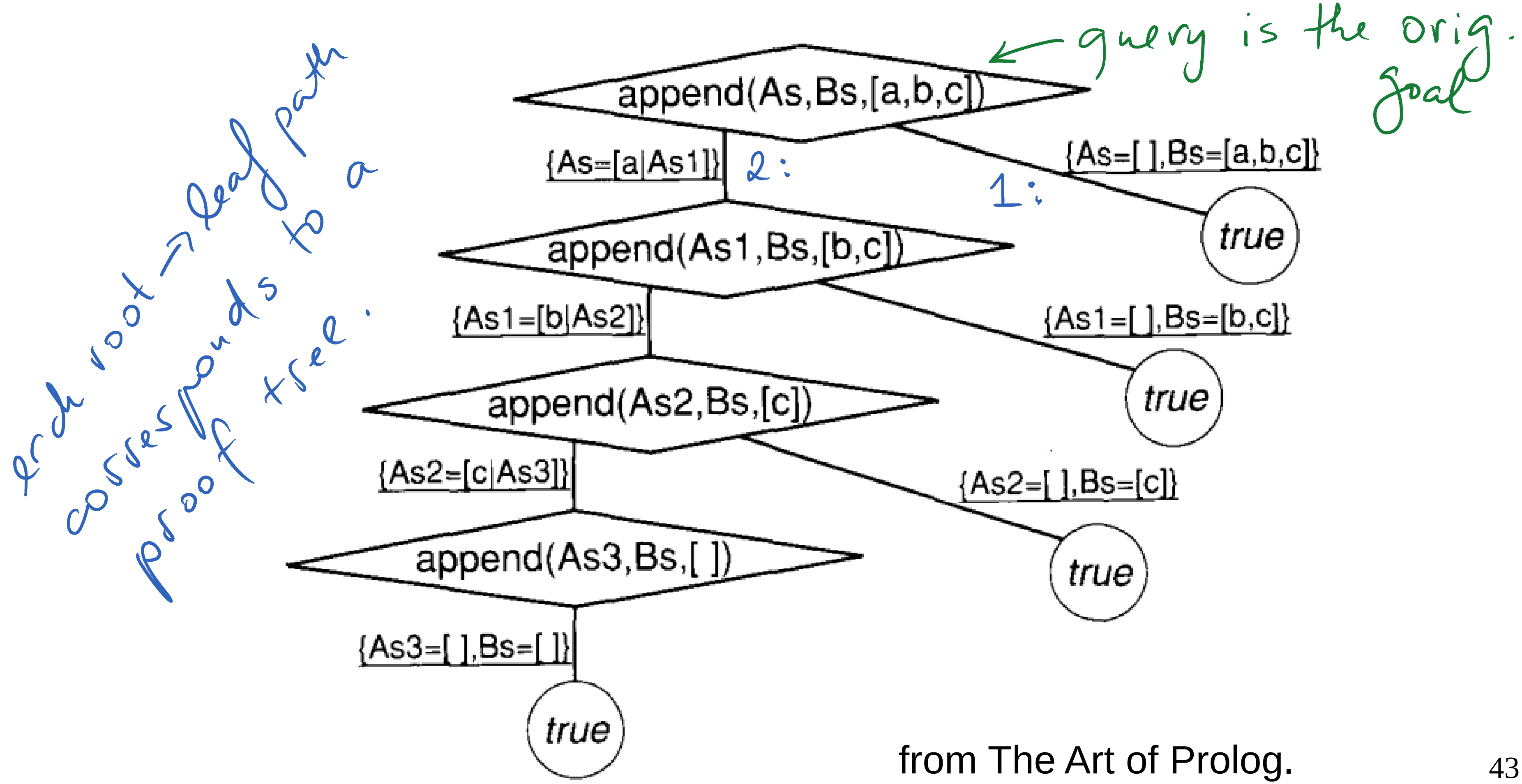
- Node is the current goal
- Edges lead to possible reductions of the goal

Number of children of a node depends on ________

your answer: number of heads matching the goal

Example search tree (for Append)

1:
```
append([],Ys,Ys).
```
2:
```
append([X|Xs],Ys,[X|Zs1) :-   append(Xs,Ys,Zs).
```



from The Art of Prolog.

Example

- Program: TODO
- Trace: holes filled in by students
- Show search tree

Algorithm

student answer:

Algorithm for (1,oracle choice)

```
solve(goal):
      match goal against a head C.H of a clause C
      if multiple matches exist: ask the oracle to pick one

      if no matching head found:
            return FAILURE
      if C has no rhs:
            return SUCCESS
      else
            solve(C.rhs)
```

Oracle is guaranteed to pick a head that is part of a proof tree

assuming a solution exists

Summary

We relied on an oracle to make just the right choice

The choice is clairvoyant: takes into consideration choices to be made by oracles down the search tree

Asking an oracle is known as non-determinism. It simplifies explanations of algorithms.

We will have to implement the oracle with backtracking in (1, backtracking)

Algorithm (n, oracular choice)

Choice of clause	1	*n*
backtracking		
by oracle	New concept**: search tree** Implementation: **ask oracle for the right choice.**	a(X) :- b(X), c(X). b(2). c(1). c(2).
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	Concept: **resolvent** Implementation: **recursion deals with reduced goals; iteration deals with rhs goals**

number of clauses on the *rhs* of rules

Analysis of this problem

Nothing too different from (1,oracle), except that we are dealing with a resolvent (ie, 2+ pending goals)

We deal with them as in (n, no choice), by first reducing the goal on top of the conceptual stack

As in (1,oracular choice), which of the alternative matches to take is up to the oracle.

What to change in (n, no choice)?

```
solve(goal):
    match goal against a head C.H of a clause C
    if multiple matches exist: ask the oracle to pick one

    if no matching head found:
        return FAILURE
    if C has no rhs:  // C is a fact
        return SUCCESS
    else    // reduce goal to the goals in the rhs of C
        for each goal in C.rhs
            if solve(goal) == FAILURE
                // oracle failed to find a solution for goal
                return FAILURE
        end for
        // goals on the rhs were solved successfully
        return SUCCESS
```

Algorithm (1, backtracking)

Choice of clause	1	*n*
backtracking	a(X) :- b(X). b(Y) :- c(Y). c(1). c(2).	
by oracle	New concept**: search tree** Implementation: **ask oracle for the right choice.**	as below, with oracular choice
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	Concept: **resolvent** Implementation: **recursion deals with reduced goals; iteration deals with rhs goals**

number of clauses on the *rhs* of rules

Implementing the oracle

We can no longer ask the oracle which of the (potentially multiple) matching heads to choose.

We need to iterate over these matches, testing whether one of them solves the goal. If we fail, we return to try the next match. This is backtracking.

Effectively, backtracking implements the oracle.

The backtracking process corresponds to dfs traversal over the search tree. See The Art of Prolog.

Algorithm for (1,backtracking)

```
solve(goal):
    for each match of goal with a head C.H of a clause C
        // this match is found with unify(), of course
        current_goal = C.rhs
        res = solve(current_goal)
        if res == SUCCESS:
            return res
    end for
    return FAILURE
```

Again, this algorithm ignores how mgus are handled. This is up to you to figure out.

Example

```
a(X) :- b(X).
b(Y) :- c(Y).
b(3).
c(1).
c(2).
?- a(Z)
```

When interpreter reaches c(1), its call stack is:

bottom

solve a(Z): matched the single a(X) head

solve b(Z): matched head b(Y); head b(3) still to explore

solve c(Z): matched head c(1); head c(2) still to explore

The implementation structure

1) *Recursion* is used to solves the new subgoal.

2) *For loop* used to iterate over alternative clauses.

Backtracking is achieved by returning to higher level of recursion and taking the next iteration of the loop.

Example executed on PA3 Prolog

```
a(X) :- b(X).
b(Y) :- c(Y).
c(1).
c(2).

a(I)?
```

Asking for solution 1
Goal: a(I)
Unify: a(X_1) and a(I)
   Unifier: {X_1->I }
Goal: b(I)
Unify: a(X_2) and b(I)
   Unifier: null
Unify: b(Y_3) and b(I)
   Unifier: {Y_3->I }
Goal: c(I)
Unify: a(X_4) and c(I)
   Unifier: null
Unify: b(Y_5) and c(I)
   Unifier: null
Unify: c(1) and c(I)
   Unifier: {I->1 }
I = 1

Asking for solution 2
Unify: c(2) and c(I)
   Unifier: {I->2 }
I = 2

Asking for solution 3
Unify: c(1) and b(I)
   Unifier: null
Unify: c(2) and b(I)
   Unifier: null
Unify: b(Y_10) and a(I)
   Unifier: null
Unify: c(1) and a(I)
   Unifier: null
Unify: c(2) and a(I)
   Unifier: null
None

Algorithm (n, backtracking)

Choice of clause	1	*n*
backtracking	Concept: backtracking is dfs of search tree. Implementation: b/tracking remembers remaining choices in a for loop on the call stack.	a(X) :- b(X), c(X). b(2). c(1). c(2).
by oracle	New concept**: search tree** Implementation: **ask oracle for the right choice.**	as below, with oracular choice
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	Concept: **resolvent** Implementation: **recursion deals with reduced goals; iteration deals with rhs goals**

number of clauses on the *rhs* of rules

Algorithm (n,backtracking) is the key task in PA3

- You will design and implement this algorithm in PA3
  - here, we provide useful hints
- Key challenge: having to deal with a resolver
  - we no longer have a single pending subgoal
- This will require a different backtracking algo design
  - one that is easier to implement with coroutines
- We will show you an outline of algo (2, backtracking)
  - you will generalize it to (n,backtracking)

Example

This example demonstrates the need to handle backtracking with coroutines:

```
a(X) :- b(X,Y), c(Y).
b(1,1).
b(2,2).
c(2).
```

2 goals on rhs

The subgoal b(X,Y) has two solutions.

- Only the second one will make c(Y) succeed.

We need a way to backtrack to the "solver" of b(X,Y)

- and ask it for the next solution

Algorithm (2, backtracking)

Restriction: we have exactly two goals on the rhs

- call them rhs[0] and rhs[1]

`solutions(goal)` returns a solution iterator

- the iterator uses `yield` to provide the next solution to `goal`

(2,backtracking):

```
for sol0 in solutions(rhs[0])
    for sol1 in solutions(rhs[1])
        if sol0 and sol1 “work together”: return SUCCESS
return FAILURE
```

Again, we are abstracting the propagation of mgus

- as a result, we need to use the informal term “goals work together”;
- it means: given mgus found in sol0, there exists a valid sol1.

Algorithm (2,backtracking), cont.

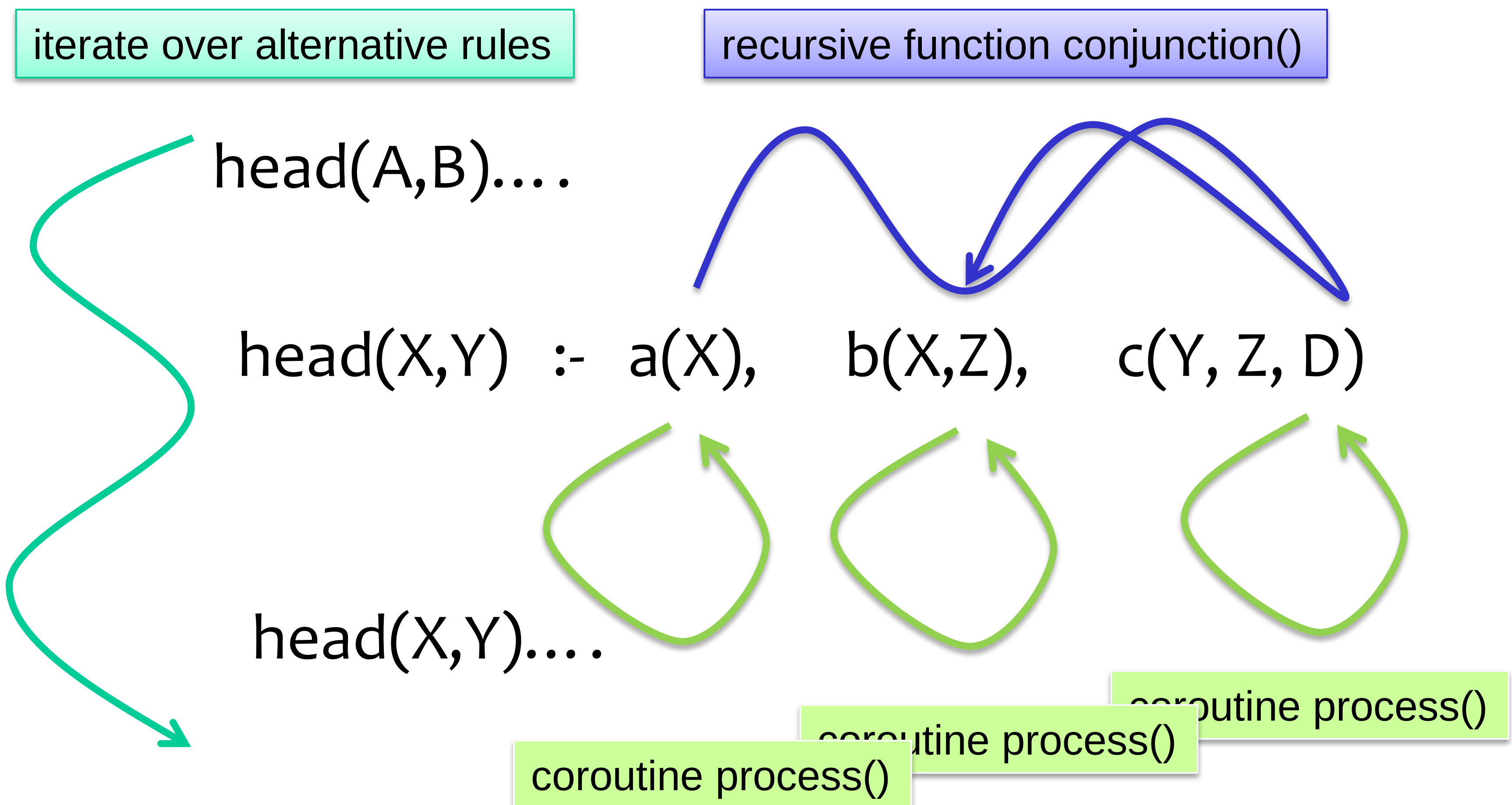
solve() must be adapted to work as a coroutine.

Key step: replace return with yield. ← familiar trick

```
solve(goal):
  for each match of goal with a head C.H of a clause C
    current_goal = C.rhs
    res = solve(current_goal)
    if res == SUCCESS:
      yield res   ~~return res~~
  return FAILURE    // think whether this needs to be yield, too
```

?

The complete view of control transfer



Example executed on PA3 Prolog

```
a(X) :- b(X), C(X).
b(2).
c(1).
c(2).

a(I)?
```

```
Asking for solution 1
Goal: a(I)
Unify: a(X_1) and a(I)
   Unifier: {X_1->I  }
Goal: b(I)
Unify: a(X_2) and b(I)
   Unifier: null
Unify: b(2) and b(I)
   Unifier: {I->2  }
Goal: c(2)
Unify: a(X_4) and c(2)
   Unifier: null
Unify: b(2) and c(2)
   Unifier: null
Unify: c(1) and c(2)
   Unifier: null
Unify: c(2) and c(2)
   Unifier: {}
I = 2
```

```
Asking for solution 2
Unify: c(1) and b(I)
   Unifier: null
Unify: c(2) and b(I)
   Unifier: null
Unify: b(2) and a(I)
   Unifier: null
Unify: c(1) and a(I)
   Unifier: null
Unify: c(2) and a(I)
   Unifier: null
None
```

Algorithm (n, backtracking)

Choice of clause	1	*n*
backtracking	Concept: backtracking is dfs of search tree. Implementation: b/tracking remembers remaining choices on the call stack.	You will design and implement this algorithm in PA3
by oracle	New concept: **search tree** Implementation: **ask oracle for the right choice.**	as below, with oracular choice
not needed	New concepts: **unifier, proof tree** Implementation: **reduce a goal and recurse**	Concept: **resolvent** Implementation: **recursion deals with reduced goals; iteration deals with rhs goals**

number of clauses on the *rhs* of rules

Reading

Required

The Art of Prolog, Chapters 4, 6, and search trees in Ch 5.

(on reserve in Kresge and in Google Books.)

Recommended

HW2: backtracking with coroutines (the regex problem)

Insightful

Logic programming via streams in CS61A textbook (SICP).