General rules in Prolog

Simple Prolog databases are not very interesting. No intelligence! We want programs that *reason*.

Recall the introductory material on back-chaining and conditional sentences:

- If X is male and X is a child
  then X is a boy.
- If X is a toddler then X is a child.

Prolog programs may (and usually will) contain such conditional sentences, which in Prolog are called general rules or clauses.

: - means "if"

Like atoms, rules in a program end with a period.

```
child(X) :- toddler(X).
```
"X is a child if X is a toddler."

```
likes(bob,X) :- child(X).
```
"Bob likes children."

```
likes(X,Y) :- baby(Y).
```
"Everyone likes babies."

Comma in a rule means "and".

```
boy(X) :- male(X), child(X).
```
"X is a boy if X is a male and X is a child."

More about clauses

For a clause

a `:-` b_1`,` ... `,` b_n`.`

- a is called the head of the clause. It must be an atom.
- $b_1,\ldots,b_n$ is called the body of the clause. Each b can be either an atom or the negation of an atom.
- read as: a if b_1 and ... and b_n.

For uniformity of terminology, an atomic sentence in Prolog is also called a clause, sometimes a unit clause. So `baby(marc)` and `likes(bob,ray)` are also clauses.

A unit clause a is said to have a as its head and an empty body. Indeed, in Prolog , you can write it as

a`.`

or as

a `:-  .`

To summarize:

> A Prolog program is a sequence of clauses, each ending with a period. A clause is a head and a body separated by `:-`. A head is an atom, and a body is a sequence of atoms or their negations separated by commas.

A program with clauses

```
baby(marc).  baby(mary).

toddler(michelle).  toddler(steve).
toddler(bob).

likes(marc,michelle).  likes(bob,ray).
likes(bob,michelle).

male(ray).  male(marc).  male(bob).
male(steve).

female(mary).  female(michelle).

child(X) :- toddler(X).

boy(X) :- male(X), child(X).
girl(X) :- female(X), child(X).

person(X) :-  male(X).
person(X) :- female(X).

adult(X) :- person(X),
    not child(X), not baby(X).

likes(X,Y) :- baby(Y), adult(X).

likes(X,bob).  /* Everyone likes bob. */

/* Comments like this may be inserted anywhere in
your program except in the middle of a clause.  They
are useful to remind you (and others) what you
intend by certain parts of your program. Use them!*/
```

Querying a general program

Queries are the same as before. But now Prolog makes use of the program's clauses to answer them.

Prolog uses back-chaining, exactly as described in the introductory material in this course.

Who likes Mary?

```
likes(X,mary).
```

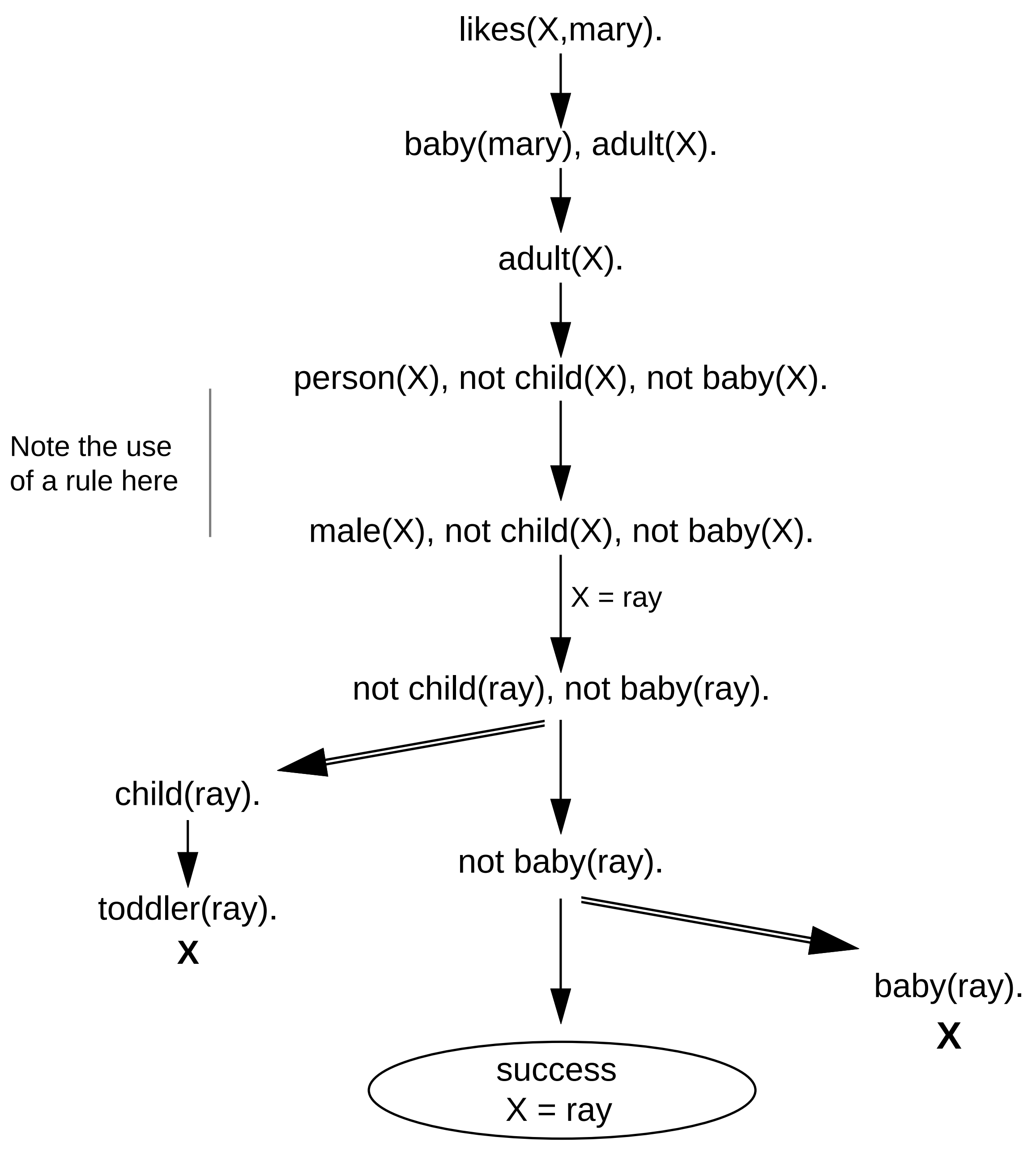
- Prolog searches in the program for the first clause which matches the query, which in this case is `likes(X,Y) :- baby(Y), adult(X).`
- So it sets up the subgoal:
  ```
      baby(mary), adult(X).
  ```
  This is a conjunctive query, and Prolog tries to answer this.
- It finds the unit clause `baby(mary)` in the program, so its new subgoal is `adult(X)`.
- It searches for the first clause in the program whose head matches this, and finds:
  ```
  adult(X) :- person(X), not child(X),
                    not baby(X).
  ```

Continued

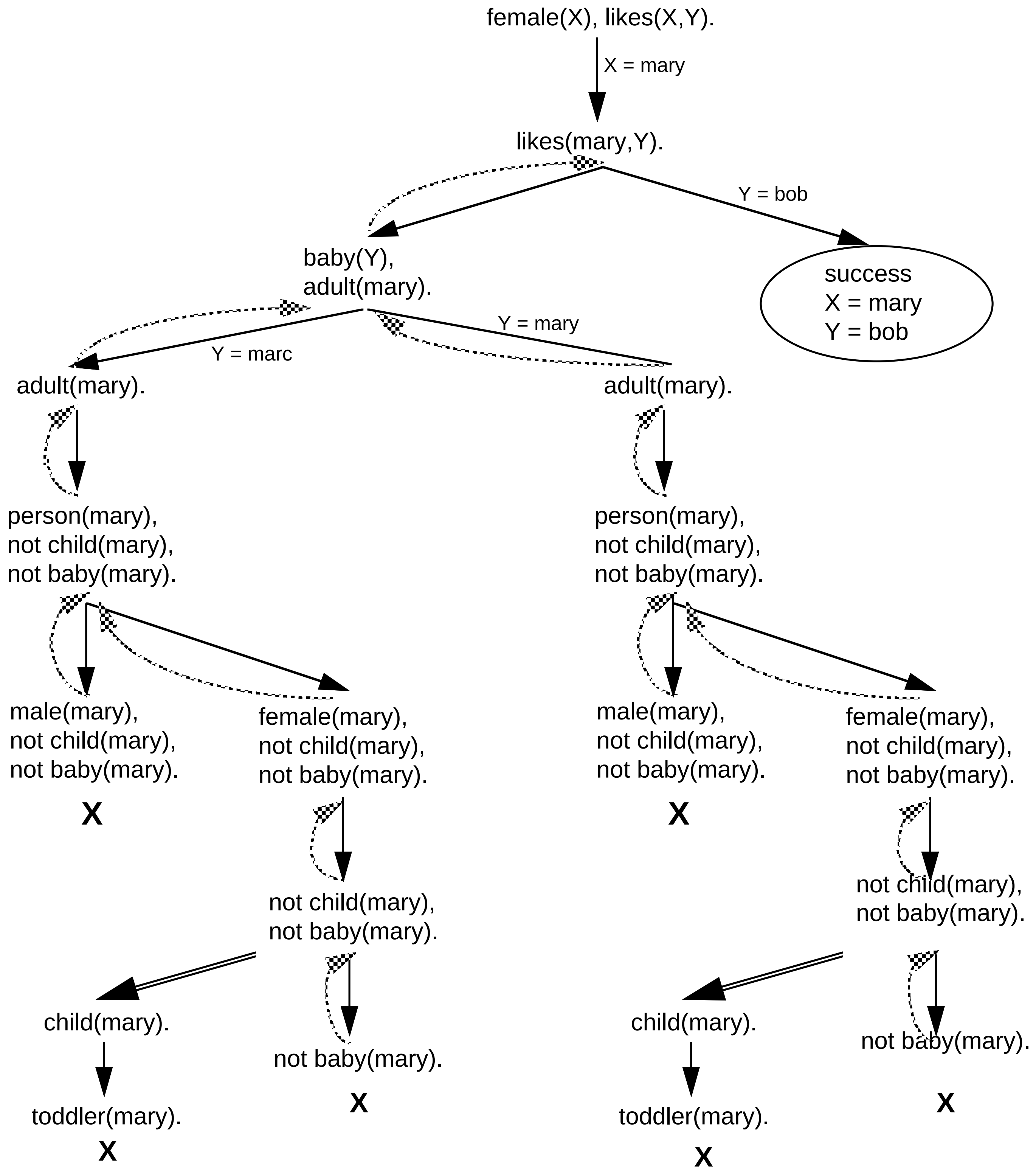
- So it establishes a new subgoal:
  ```
  person(X),
       not child(X), not baby(X).
  ```
- Again, it searches for the first clause whose head matches `person(X)`, which is:
  `person(X) :-  male(X).`
- So it establishes the new subgoal:
  ```
  male(X), not child(X), not baby(X).
  ```
- The first clause in the program which matches `male(X)` is `male(ray)`, so the next subgoal is:
  ```
  not child(ray), not baby(ray).
  ```
- Prolog works on `not child(ray)`.
  - It attempts `child(ray)`. This leads to trying to solve `toddler(ray)` which fails.
  - Therefore, `not child(ray)` succeeds.
- Prolog now works on `not baby(ray)` which eventually succeeds.

So Prolog succeeds on the top level goal `likes(X,mary)` with `X = ray`.

Evaluating `likes(X,mary).`


likes(X,mary).
baby(mary), adult(X).
adult(X).
person(X), not child(X), not baby(X).
Note the use of a rule here
male(X), not child(X), not baby(X).
X = ray
not child(ray), not baby(ray).
child(ray).
toddler(ray).
X
not baby(ray).
baby(ray).
X
success
X = ray

Backtracking



Writing meaningful Prolog

When writing Prolog programs, it is important to consider carefully what we intend the predicates to *mean*, and to make sure that the clauses we write are all *true*.

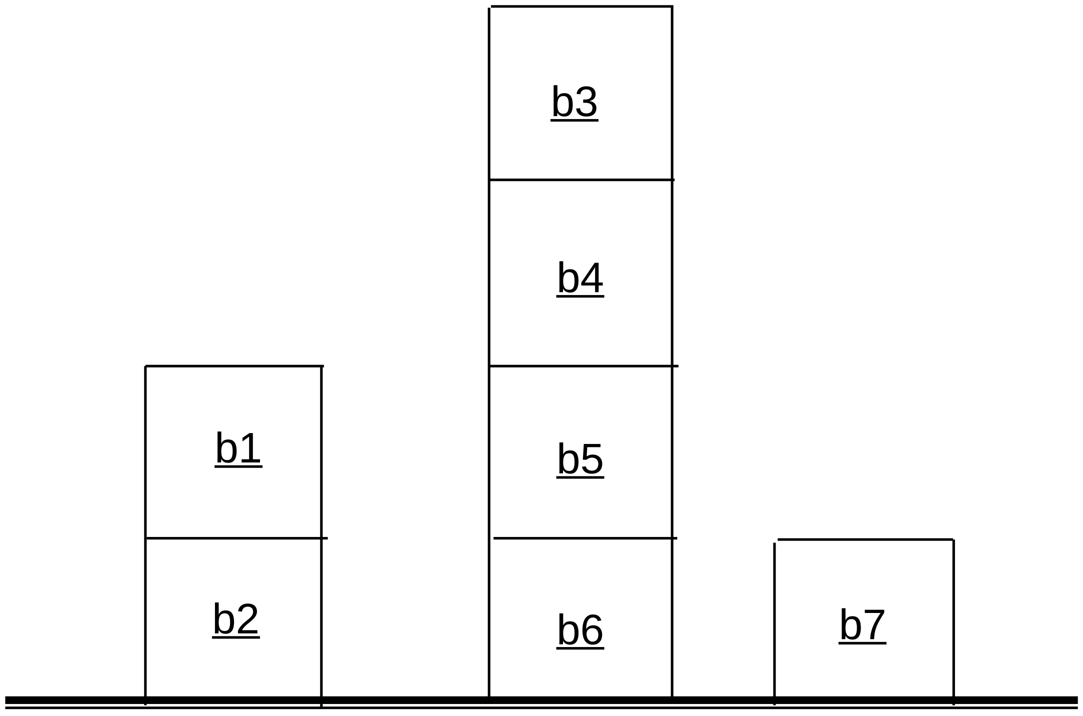
The next issue to consider is whether or not we have included enough clauses to allow Prolog to draw appropriate conclusions involving the predicates.

Finally, we need to consider how back-chaining will use the clauses when actually deriving conlusions.

To summarize, we need

1. the truth, and nothing but;
2. the whole truth;
3. presented in the right form for back-chaining.

A blocks world



Would like to describe the scene and get Prolog to deduce that

Block 3 is above Block 6

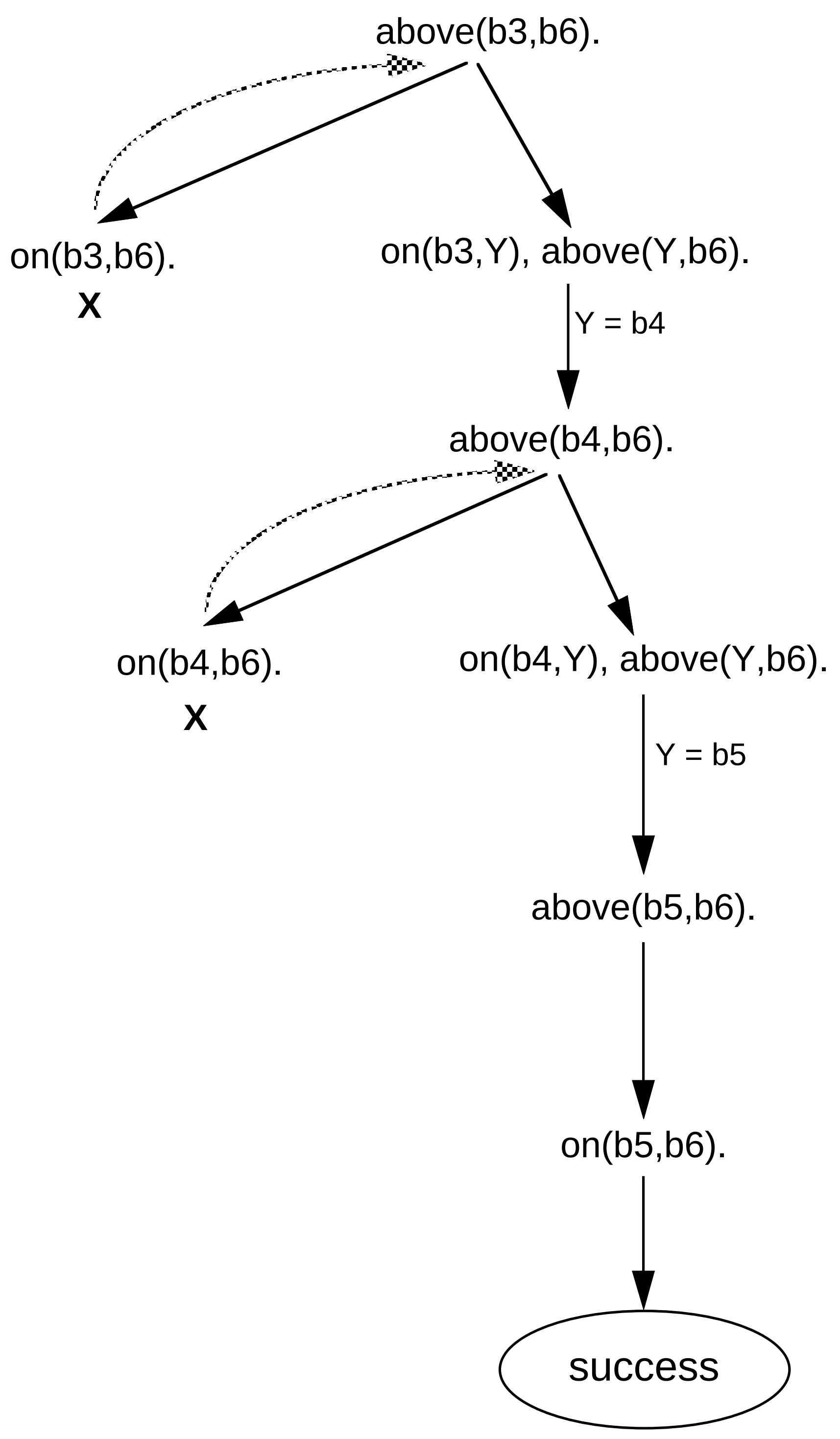
Block 1 is to the left of Block 7

Block 4 is to the right of Block 2

A blocks world program

```
/*  on(X,Y) means that block X is directly on top
    of (touching) block Y. */

on(b1,b2).  on(b3,b4).  on(b4,b5).  on(b5,b6).

/* just_left(X,Y) means that block X is on the
   table and is immediately to the left of block Y,
   which is also on the table.  */

just_left(b2,b6).  just_left(b6,b7).

/* above(X,Y) means that block X is somewhere above
   Y in the pile of blocks in which Y occurs.  */

above(X,Y) :- on(X,Y).
above(X,Z) :- on(X,Y), above(Y,Z).

/* left(X,Y) means that block X is to be found
   somewhere to the left of block Y.
   Thus, left(b2,b7), left(b1,b7) and left(b1,b4)
   are all true.  left(b5,b1) is false.  */

left(X,Z) :- above(X,Y), left(Y,Z).  /* X is high */
left(X,Y) :- above(Y,Z), left(X,Z).  /* X is low */
left(X,Y) :- just_left(X,Y).
left(X,Y) :- just_left(X,Z), left(Z,Y).

right(X,Y) :- left(Y,X).
```

Evaluating `above(b3,b6).`



Recursion

The example we just saw makes use of recursion.

Recursive clause = one in which the predicate of the head is the same as a predicate mentioned in the body.

Example:

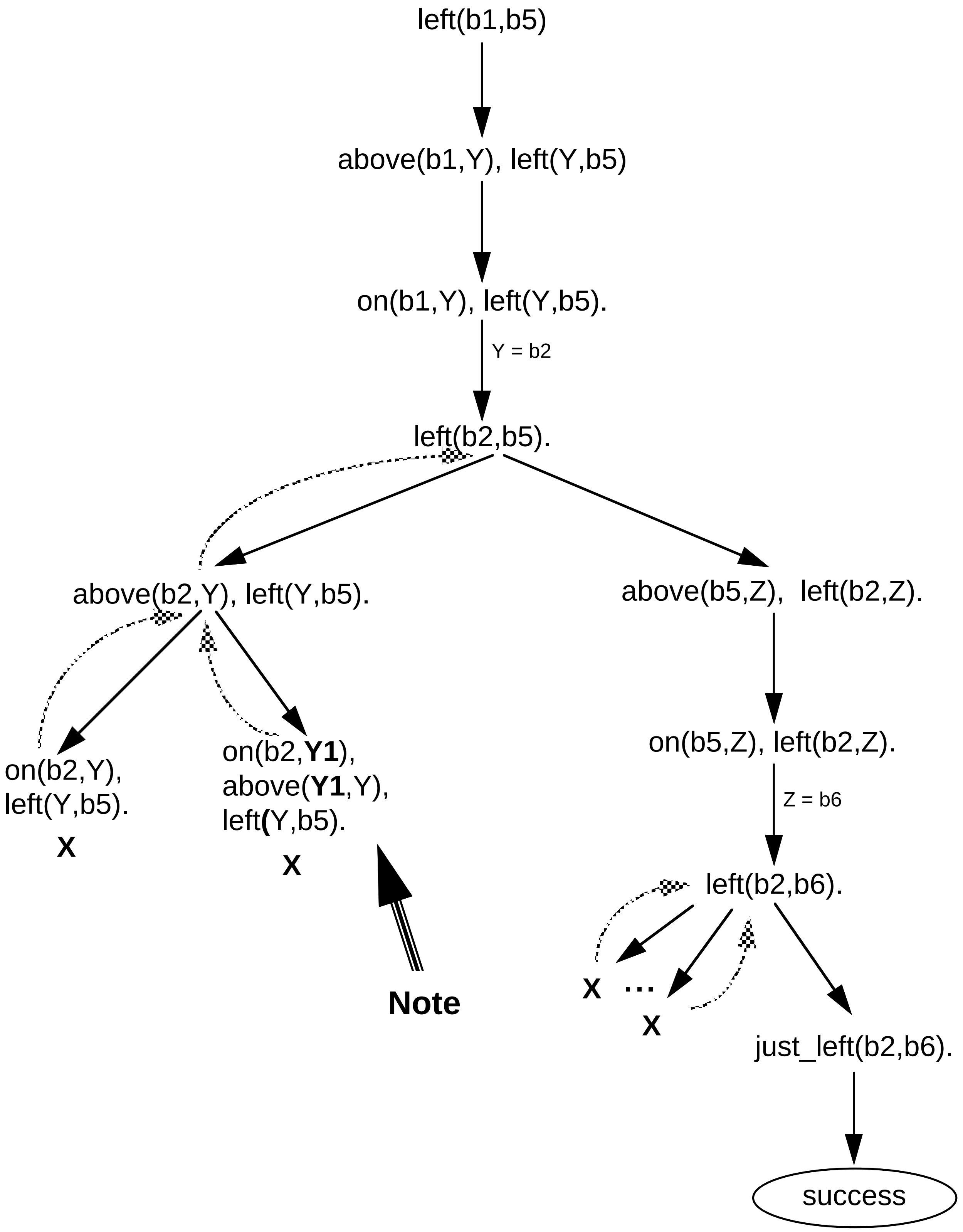
```
above(X,Z) :- on(X,Y), above(Y,Z).
```

If *x* is on *y* and *y* is above *z*, then *x* is above *z*

Most modern programming languages (Pascal, C) provide recursion, which is usually taught as an advanced technique.

In fact, it’s really quite a simple idea and lies at the heart of Prolog programming.

Evaluating `left(b1,b5).`



Renaming variables

Sometimes, as in the previous example, it is necessary to rename variables in a program clause before using them to generate a new subgoal .

In the previous example, we had a subgoal:

```
      above(b2,Y), left(Y,b5).
```

We were generating a new subgoal from this one by using the program clause

```
above(X,Z) :- on(X,Y), above(Y,Z).
```

Notice that the subgoal and the program clause have a variable, Y, in common.

Suppose we just blindly generate a new subgoal:

```
      on(b2,Y), above(Y,Y), left(Y,b5)
```

which is not right.

But if we rename the variables of the program clause so that they are different from the variables of the subgoal (after all, they can have any names we want):

```
above(X1,Z1) :- on(X1,Y1), above(Y1,Z1).
```

Then the new subgoal is:

```
      on(b2,Y1), above(Y1,Y), left(Y,b5)
```

which works fine.

The moral

When you (as opposed to Prolog) are using a program clause and a subgoal to establish a new subgoal, make sure the clause's variables are different from the subgoal's variables.

In fact, Prolog does this automatically by storing clauses using internal variables which it guarantees to be different from any other variable in any other program clause or query.

Example: Recall the program clause

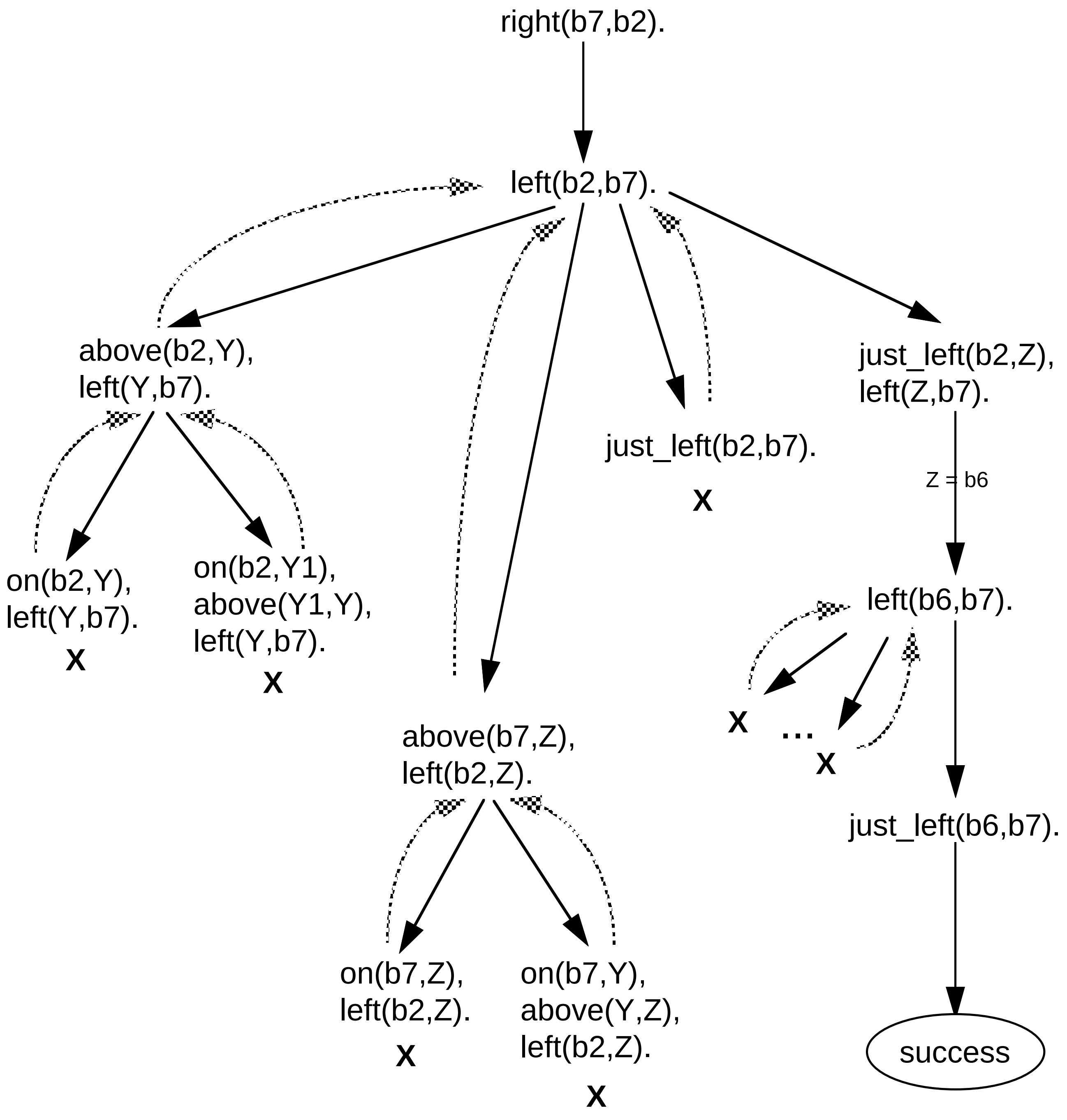
```
     likes(X,bob).  /* Everyone likes Bob */
```

Suppose this is the entire program and consider the query "Who likes Bob?":

```
     likes(Y,bob).
     Y = _967.
     yes
```

`_967` here is an internal variable which Prolog used to replace the variable `X` in the original program clause. The answer `Y = _967` means that anything at all is an example of a Y that likes Bob.

Evaluating `right(b7,b2).`



Nonterminating Programs

When using recursion, it is possible to write programs which go on forever. For example:

```
left(X,Z) :- left(X,Y), just_left(Y,Z).
```

What this says is *correct:* If x is to the left of y and y is just to the left of z, then x is to the left of z.

However:

$$\begin{tikzcd}
\text{left(b1,b2)} \arrow[d] \\
\text{left(b1,Y), just\_left(Y,b2)} \arrow[d] \\
\text{left(b1,Y1), just\_left(Y1,Y), just\_left(Y,b2)} \arrow[d] \\
\text{left(b1,Y2), just\_left(Y2,Y1), just\_left(Y1,Y), just\_left(Y,b2)} \arrow[d] \\
\dots
\end{tikzcd}$$

Eventually, Prolog reports: “Sorry, there isn’t enough memory to complete evaluation of the query.”

Avoiding non-termination

When writing recursive programs, there is no simple way to guarantee that they will terminate.

However, a good rule of thumb is that when a clause is recursive, the body should contain at least one atom before the recursive one to provide a new value for one of the variables.

For example, instead of

```
    left(X,Z) :- left(X,Y), just_left(Y,Z).
```

we should write

```
    left(X,Z) :- just_left(Y,Z), left(X,Y).
```

find a value for Y

These two clauses mean the same thing in English, but because of the left-to-right order of Prolog, the second one will reach the recursive predicate only after we have found a specific value for Y.