Prolog Programming

slides written by

Dr W.F. Clocksin

The Plan

- An example program
- Syntax of terms
- Some simple programs
- Terms as data structures, unification
- The Cut
- Writing real programs

What is Prolog?

- Prolog is the most widely used language to have been inspired by logic programming research. Some features:
- Prolog uses logical variables. These are not the same as variables in other languages. Programmers can use them as ‘holes’ in data structures that are gradually filled in as computation proceeds.

…More

- Unification is a built-in term-manipulation method that passes parameters, returns results, selects and constructs data structures.
- Basic control flow model is backtracking.
- Program clauses and data have the same form.
- The relational form of procedures makes it possible to define ‘reversible’ procedures.

…More

- Clauses provide a convenient way to express case analysis and nondeterminism.
- Sometimes it is necessary to use control features that are not part of ‘logic’.
- A Prolog program can also be seen as a relational database containing rules as well as facts.

What a program looks like

```
/* At the Zoo */

elephant(george).
elephant(mary).

panda(chi_chi).
panda(ming_ming).

dangerous(X) :- big_teeth(X).
dangerous(X) :- venomous(X).

guess(X, tiger) :- stripey(X), big_teeth(X), isaCat(X).
guess(X, koala) :- arboreal(X), sleepy(X).
guess(X, zebra) :- stripey(X), isaHorse(X).
```

Prolog is a ‘declarative’ language

- Clauses are statements about what is true about a problem, instead of instructions how to accomplish the solution.
- The Prolog system uses the clauses to work out how to accomplish the solution by searching through the space of possible solutions.
- Not all problems have pure declarative specifications. Sometimes extralogical statements are needed.

Example: Concatenate lists a and b

In an imperative language

```
list procedure cat(list a, list b)
{
   list t = list u = copylist(a);
   while (t.tail != nil) t = t.tail;
   t.tail = b;
   return u;
}
```

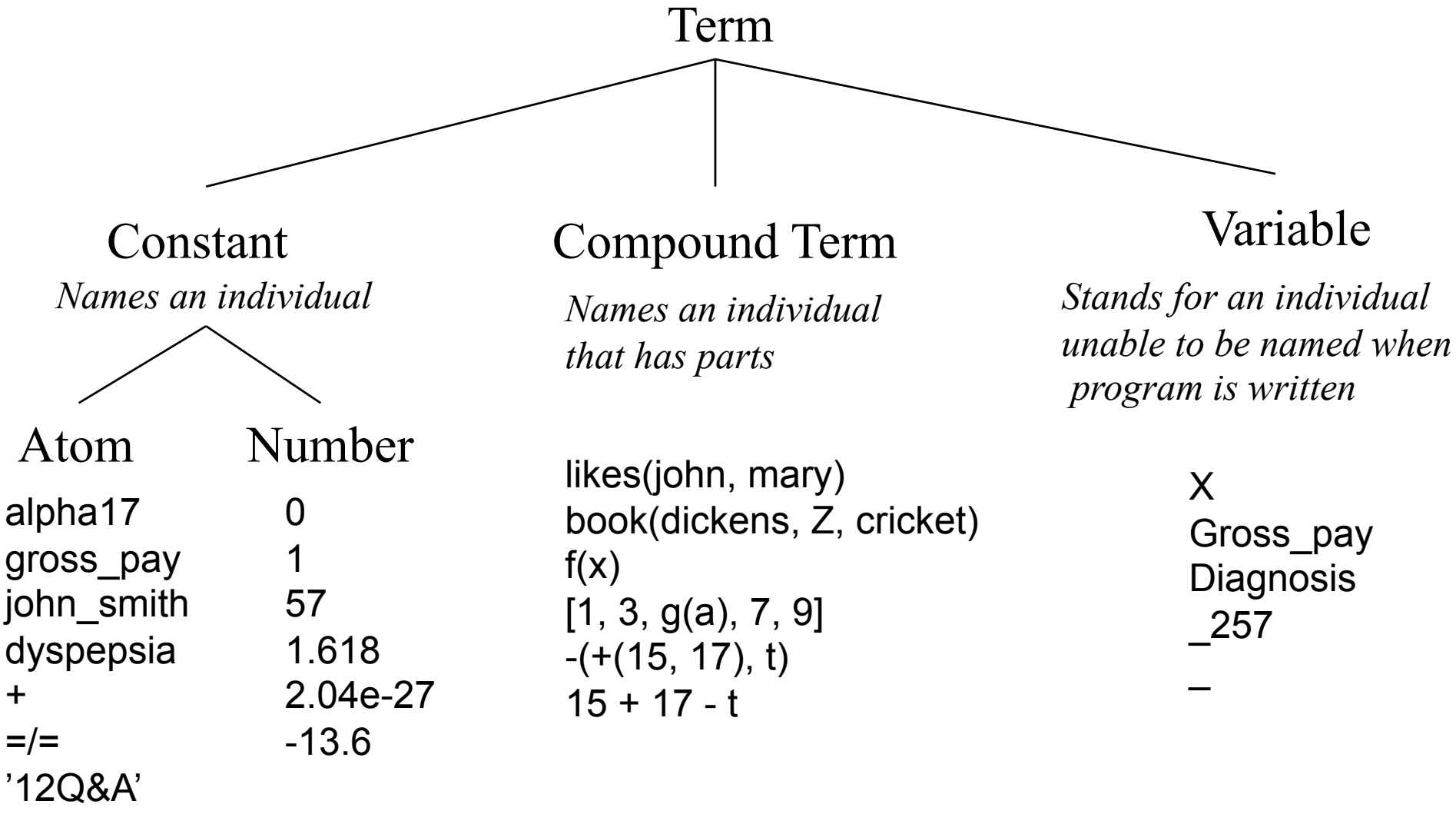
In a functional language

```
cat(a,b) ≡
 if b = nil then a
else cons(head(a), cat(tail(a),b))
```

In a declarative language

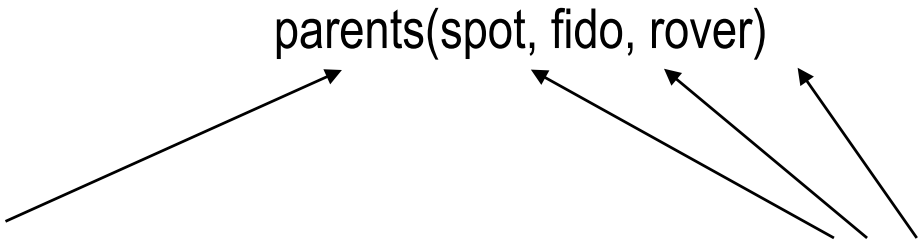
```
cat([], Z, Z).
cat([H|T], L, [H|Z]) :- cat(T, L, Z).
```

Complete Syntax of Terms



Compound Terms

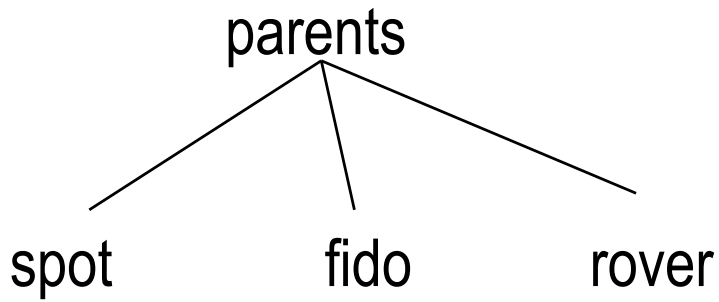
The parents of Spot are Fido and Rover.



Functor (an atom) of arity 3.

components (any terms)

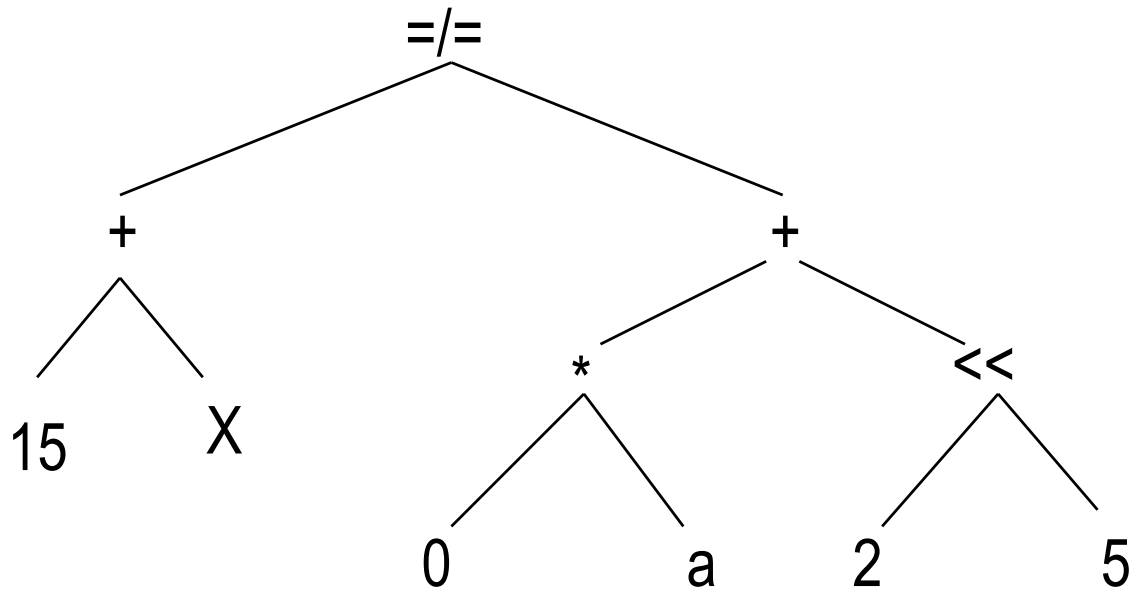
It is possible to depict the term as a tree:



Compound Terms

Some atoms have built-in operator declarations so they may be written in a syntactically convenient form. The meaning is not affected. This example looks like an arithmetic expression, but might not be. It is just a term.

=/=(15+X, (0*a)+(2<<5))



More about operators

- Any atom may be designated an operator. The only purpose is for convenience; the only effect is how the term containing the atom is parsed. Operators are ‘syntactic sugar’.
- We won’t be designating operators in this course, but it is as well to understand them, because a number of atoms have built-in designations as operators.
- Operators have three properties: position, precedence and associativity.

more…

Examples of operator properties

Position	Operator Syntax	Normal Syntax
Prefix:	-2	-(2)
Infix:	5+17	+(17,5)
Postfix:	N!	!(N)

Associativity: left, right, none.

X+Y+Z is parsed as (X+Y)+Z

because addition is left-associative.

These are all the same as the normal rules of arithmetic.

Precedence: an integer.

X+Y*Z is parsed as X+(Y*Z)

because multiplication has higher precedence.

The last point about Compound Terms…

Constants are simply compound terms of arity 0.

badger

means the same as

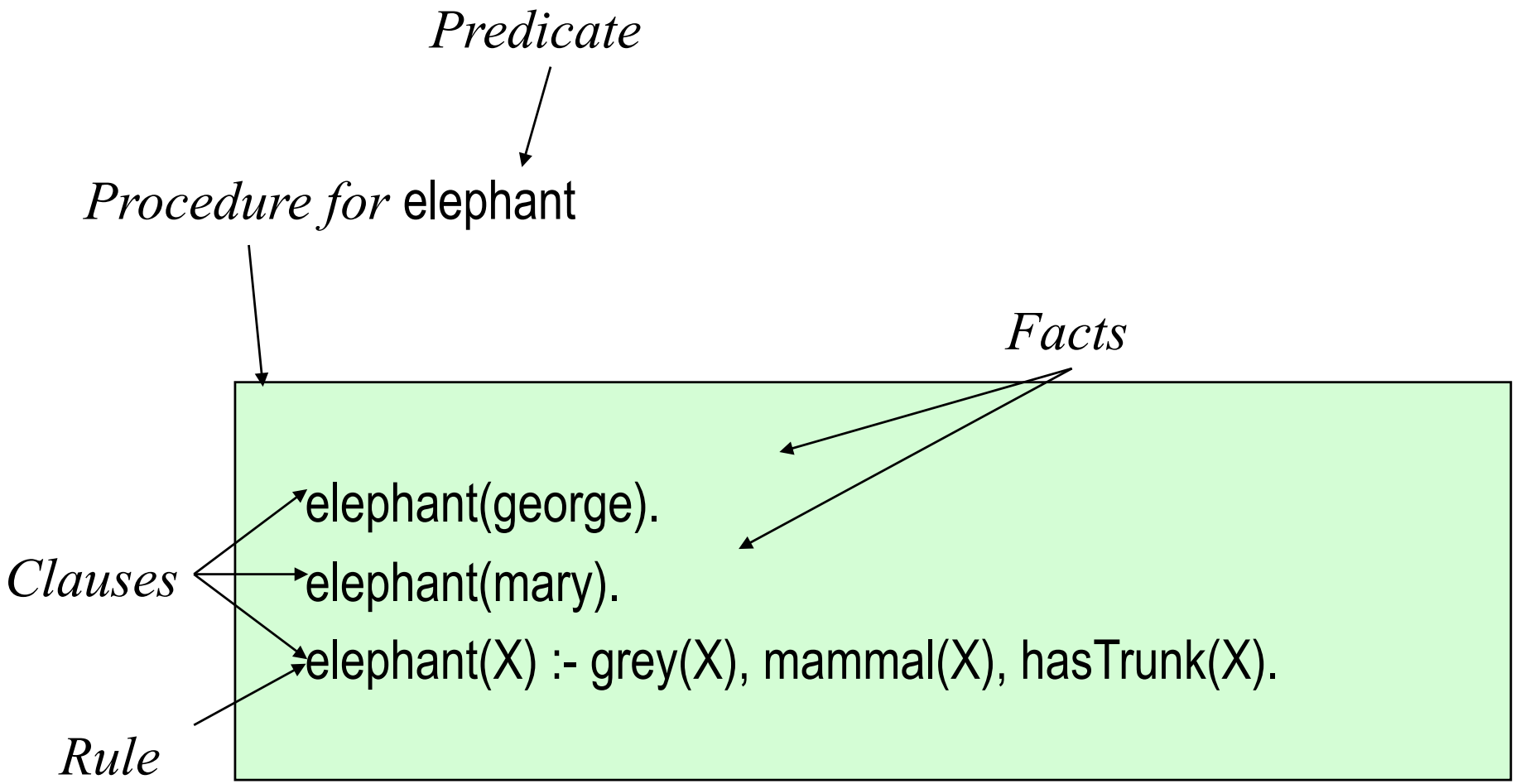
badger()

Structure of Programs

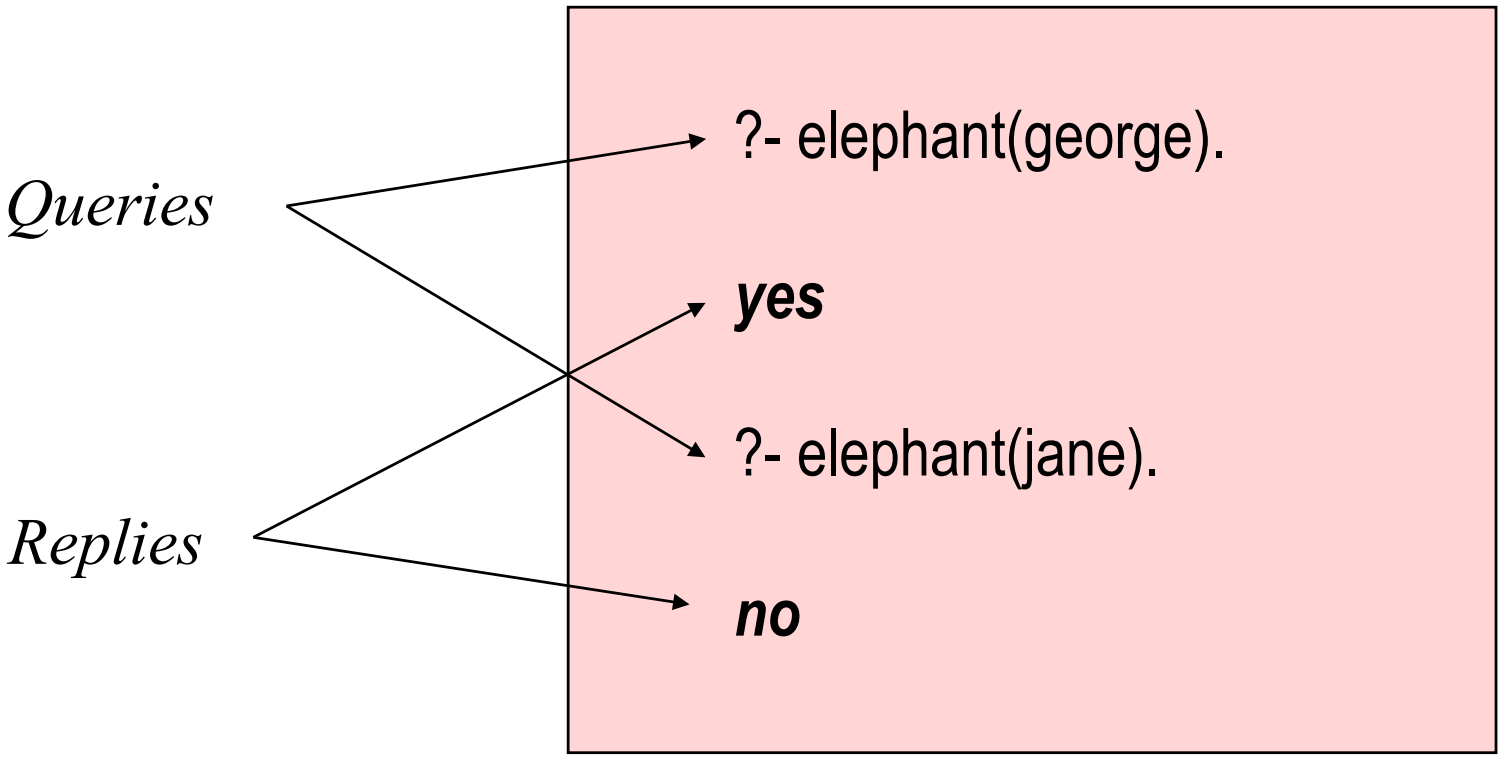
- Programs consist of procedures.
- Procedures consist of clauses.
- Each clause is a fact or a rule.
- Programs are executed by posing queries.

An example…

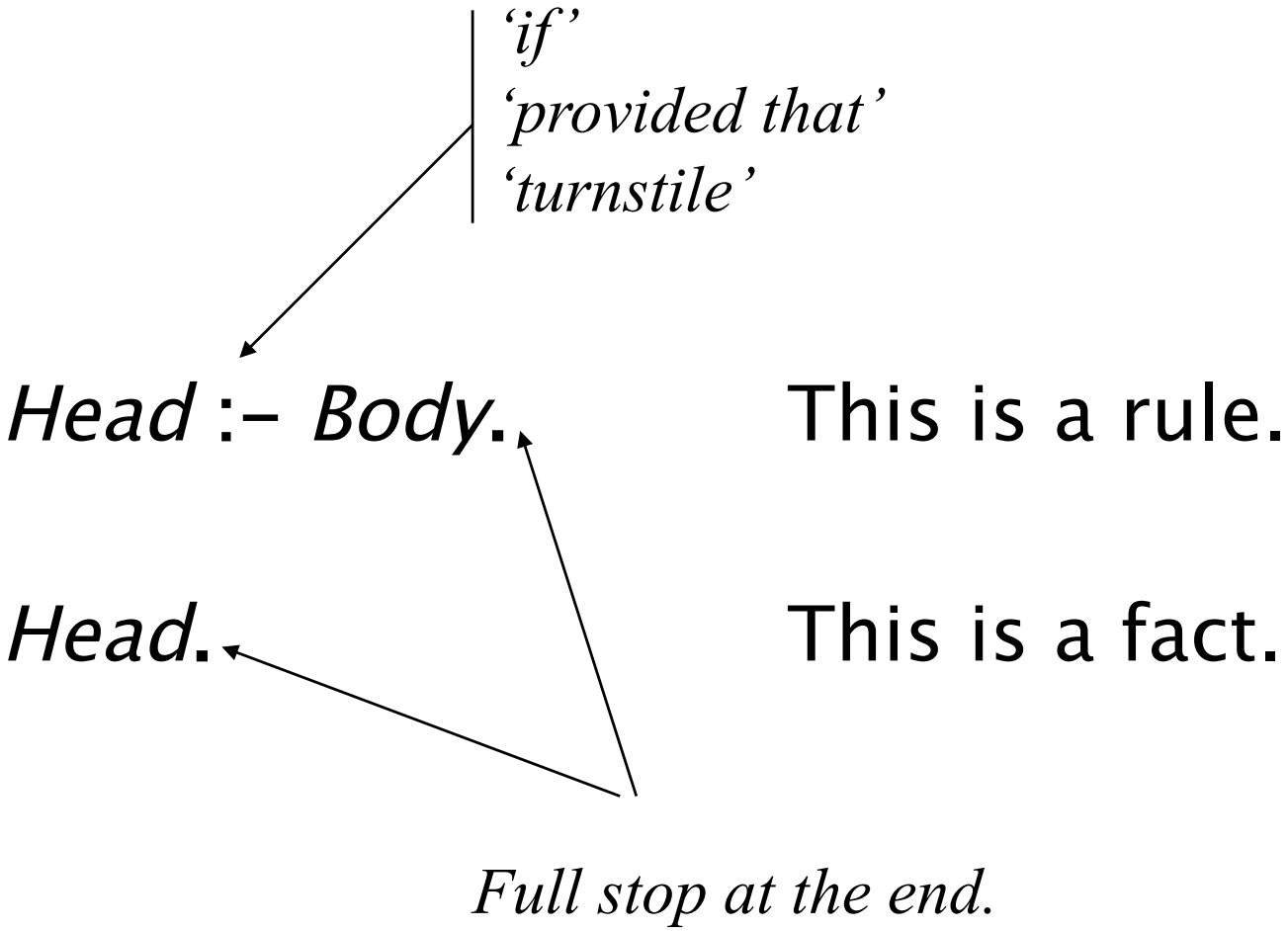
Example


Predicate
Procedure for elephant
Facts
elephant(george).
elephant(mary).
elephant(X) :- grey(X), mammal(X), hasTrunk(X).
Clauses
Rule

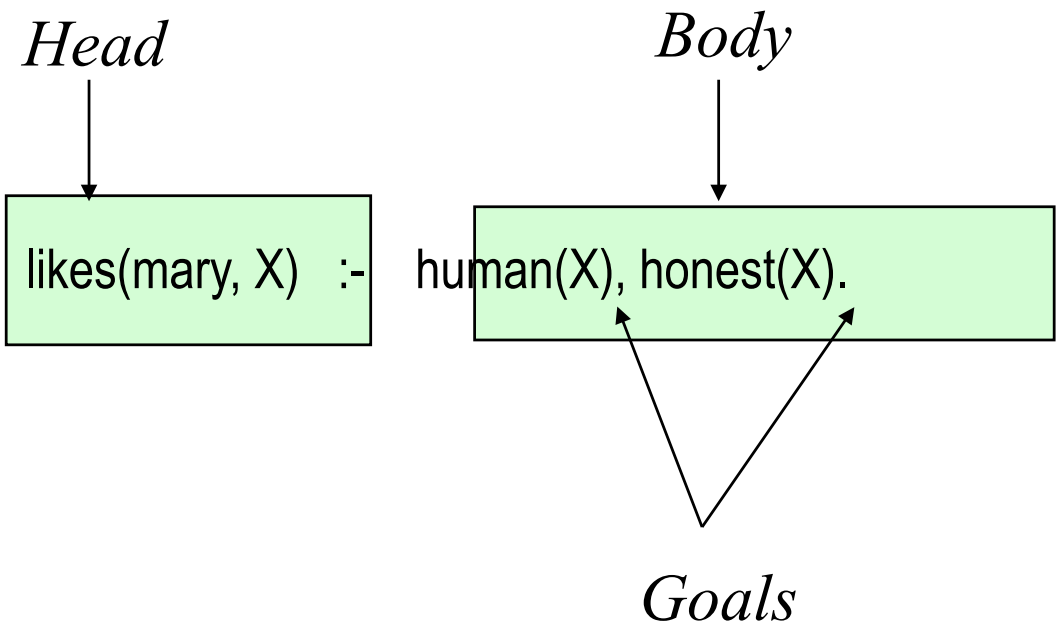
Example



Clauses: Facts and Rules



Body of a (rule) clause contains goals.



Exercise: Identify all the parts of Prolog text you have seen so far.

Interpretation of Clauses

Clauses can be given a declarative reading or a procedural reading.

Form of clause: H :- G_1, G_2, …, G_n.

Declarative reading: "That H is provable follows from goals G_1, G_2, …, G_n being provable."

Procedural reading: "To execute procedure H, the procedures called by goals G_1, G_2, …, G_n are executed first."

```
male(bertram).
male(percival).

female(lucinda).
female(camilla).

pair(X, Y) :- male(X), female(Y).
```

```
?- pair(percival, X).
?- pair(apollo, daphne).
?- pair(camilla, X).
?- pair(X, lucinda).
?- pair(X, X).
?- pair(bertram, lucinda).
?- pair(X, daphne).
?- pair(X, Y).
```

Worksheet 2

```
drinks(john, martini).
drinks(mary, gin).
drinks(susan, vodka).
drinks(john, gin).
drinks(fred, gin).

pair(X, Y, Z) :-
        drinks(X, Z),
        drinks(Y, Z).
```

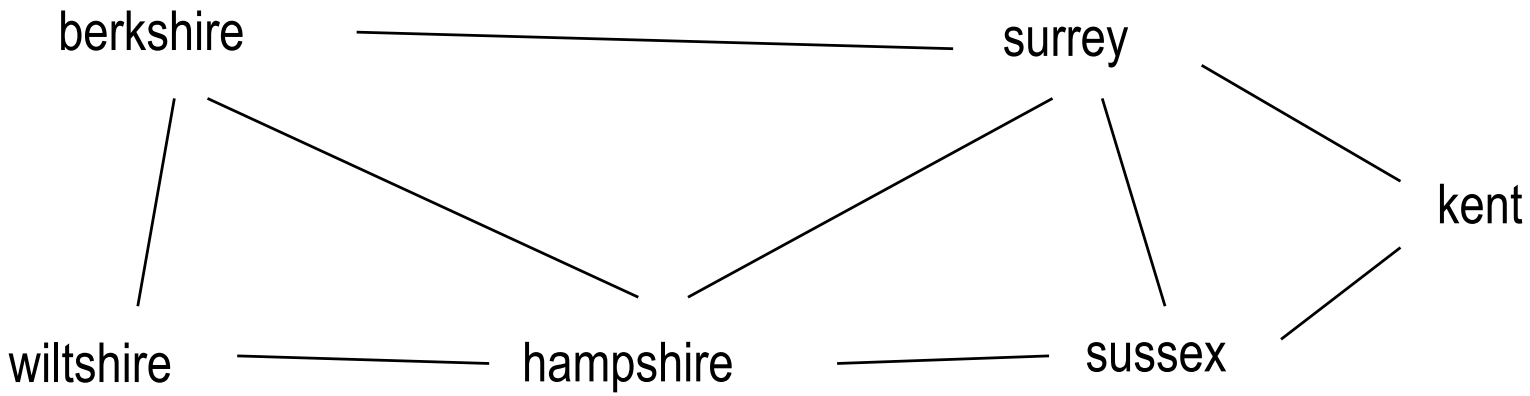
```
?- pair(X, john, martini).
?- pair(mary, susan, gin).
?- pair(john, mary, gin).
?- pair(john, john, gin).
?- pair(X, Y, gin).
?- pair(bertram, lucinda).
?- pair(bertram, lucinda, vodka).
?- pair(X, Y, Z).
```

This definition forces X and Y to be distinct:

```
pair(X, Y, Z) :- drinks(X, Z), drinks(Y, Z), X \== Y.
```

Worksheet 3

(a) Representing a symmetric relation.
(b) Implementing a strange ticket condition.



How to represent this relation?
Note that borders are symmetric.

WS3

This relation represents one ‘direction’ of border:

```
border(sussex, kent).
border(sussex, surrey).
border(surrey, kent).
border(hampshire, sussex).
border(hampshire, surrey).
border(hampshire, berkshire).
border(berkshire, surrey).
border(wiltshire, hampshire).
border(wiltshire, berkshire).
```

What about the other?

(a) Say

```
border(kent, sussex).
border(sussex, kent).
```

⋮

(b) Say

```
adjacent(X, Y) :- border(X, Y).
adjacent(X, Y) :- border(Y, X).
```

(c) Say

```
border(X, Y) :- border(Y, X).
```

WS3

Now a somewhat strange type of discount ticket. For the ticket to be valid, one must pass through an intermediate county.

A valid ticket between a start and end county obeys the following rule:

```
valid(X, Y) :- adjacent(X, Z), adjacent(Z, Y)
```

WS3

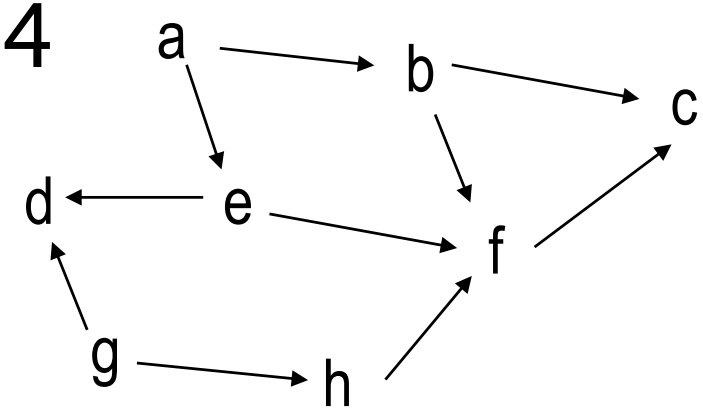
```
border(sussex, kent).
border(sussex, surrey).
border(surrey, kent).
border(hampshire, sussex).
border(hampshire, surrey).
border(hampshire, berkshire).
border(berkshire, surrey).
border(wiltshire, hampshire).
border(wiltshire, berkshire).

adjacent(X, Y) :- border(X, Y).
adjacent(X, Y) :- border(Y, X).
```

```
valid(X, Y) :-
        adjacent(X, Z),
        adjacent(Z, Y)
```

```
?- valid(wiltshire, sussex).
?- valid(wiltshire, kent).
?- valid(hampshire, hampshire).
?- valid(X, kent).
?- valid(sussex, X).
?- valid(X, Y).
```

Worksheet 4



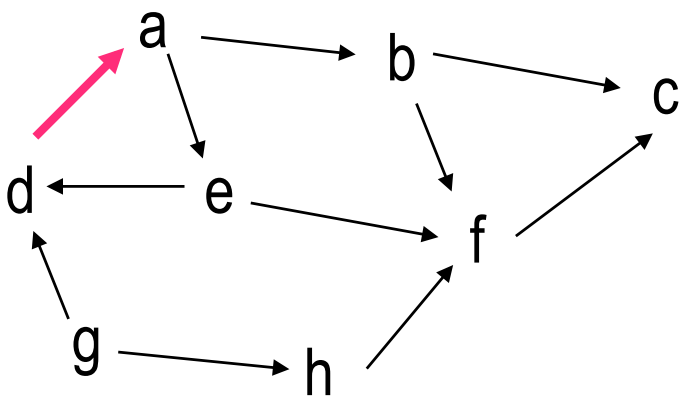

```
a(g, h).
a(g, d).
a(e, d).
a(h, f).
a(e, f).
a(a, e).
a(a, b).
a(b, f).
a(b, c).
a(f, c).
```

Note that Prolog can distinguish between the 0-ary constant a *(the name of a node) and the 2-ary functor* a *(the name of a relation).*

```
path(X, X).
path(X, Y) :- a(X, Z), path(Z, Y).
```

```
?- path(f, f).
?- path(a, c).
?- path(g, e).
?- path(g, X).
?- path(X, h).
```

But what happens if…




```
a(g, h).
a(g, d).
a(e, d).
a(h, f).
a(e, f).
a(a, e).
a(a, b).
a(b, f).
a(b, c).
a(f, c).
a(d, a).
```

```
path(X, X).
path(X, Y) :- a(X, Z), path(Z, Y).
```

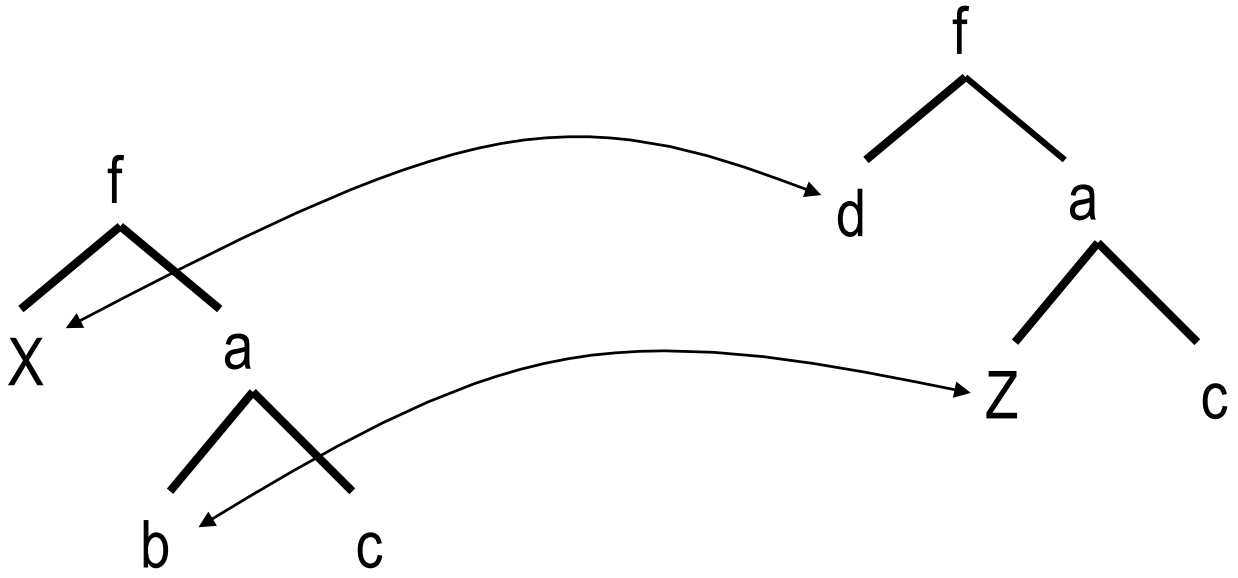
This program works only for acyclic graphs. The program may infinitely loop given a cyclic graph. We need to leave a ‘trail’ of visited nodes. This is accomplished with a data structure (to be seen later).

Unification

- Two terms unify if substitutions can be made for any variables in the terms so that the terms are made identical. If no such substitution exists, the terms do not unify.
- The Unification Algorithm proceeds by recursive descent of the two terms.
  - Constants unify if they are identical
  - Variables unify with any term, including other variables
  - Compound terms unify if their functors and components unify.

Examples

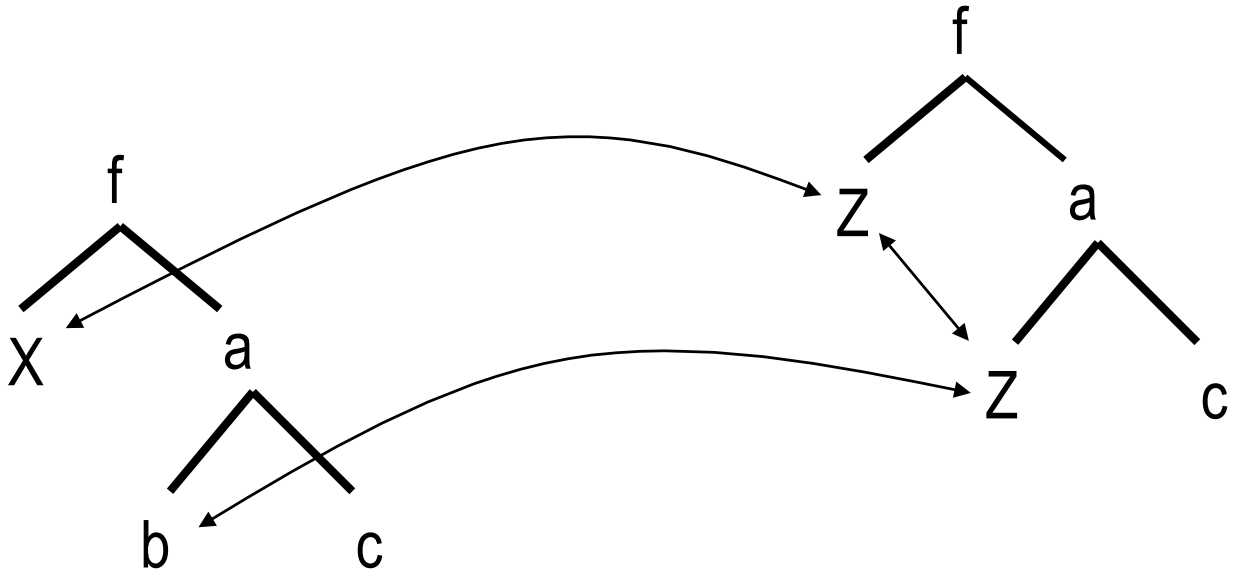
The terms f(X, a(b,c)) and f(d, a(Z, c)) unify.



The terms are made equal if d is substituted for X, and b is substituted for Z. We also say X is instantiated to d and Z is instantiated to b, or X/d, Z/b.

Examples

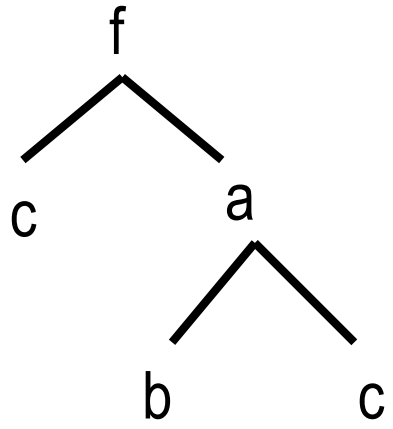
The terms f(X, a(b,c)) and f(Z, a(Z, c)) unify.

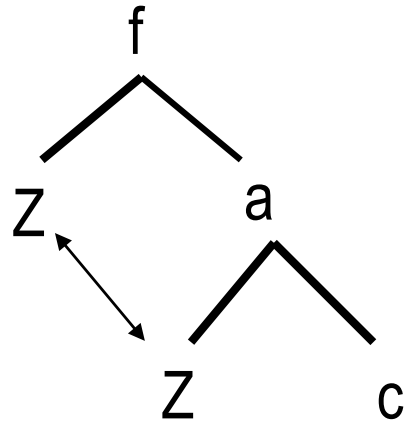


Note that Z co-refers within the term.
Here, X/b, Z/b.

Examples

The terms f(c, a(b,c)) and f(Z, a(Z, c)) do not unify.

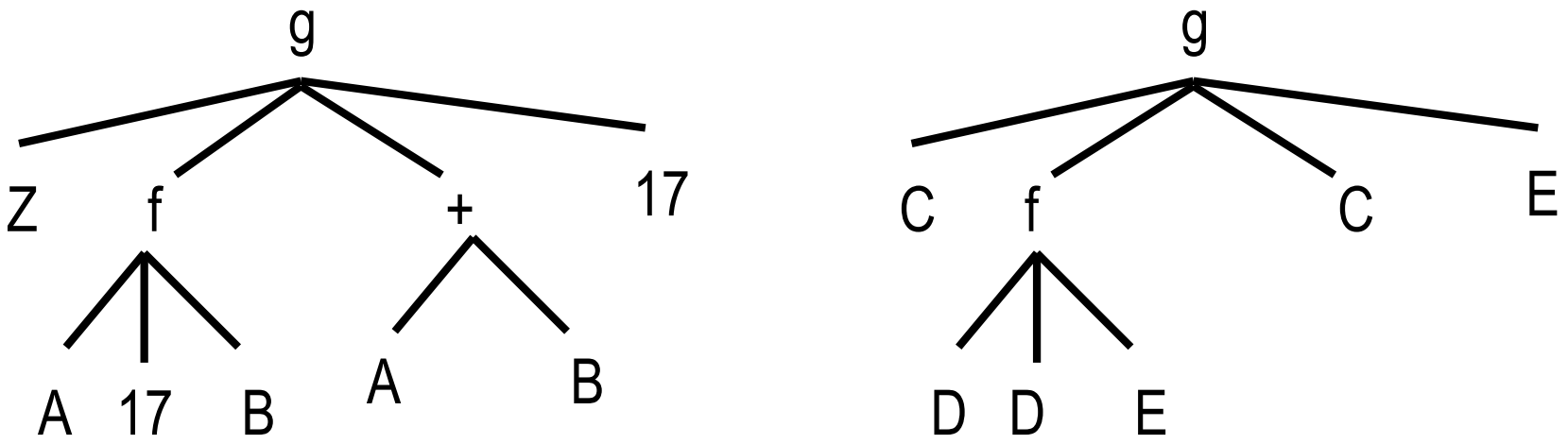




No matter how hard you try, these two terms cannot be made identical by substituting terms for variables.

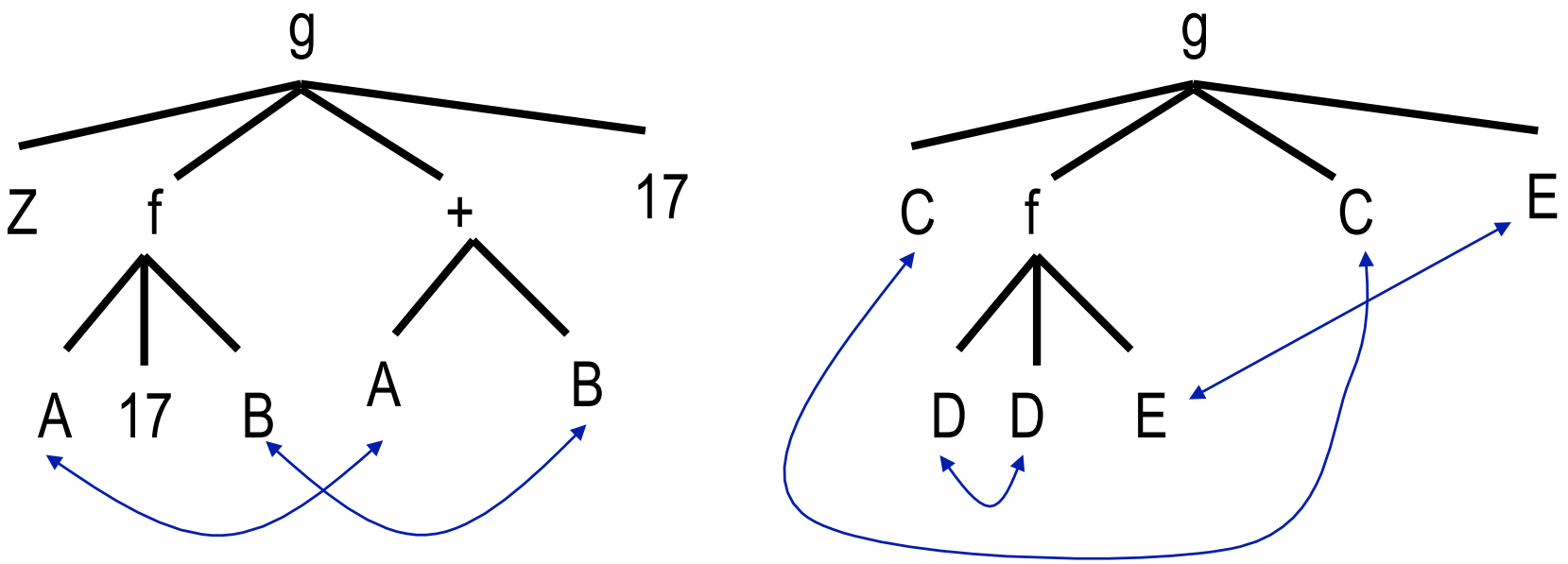
Exercise

Do terms g(Z, f(A, 17, B), A+B, 17) and g(C, f(D, D, E), C, E) unify?



Exercise

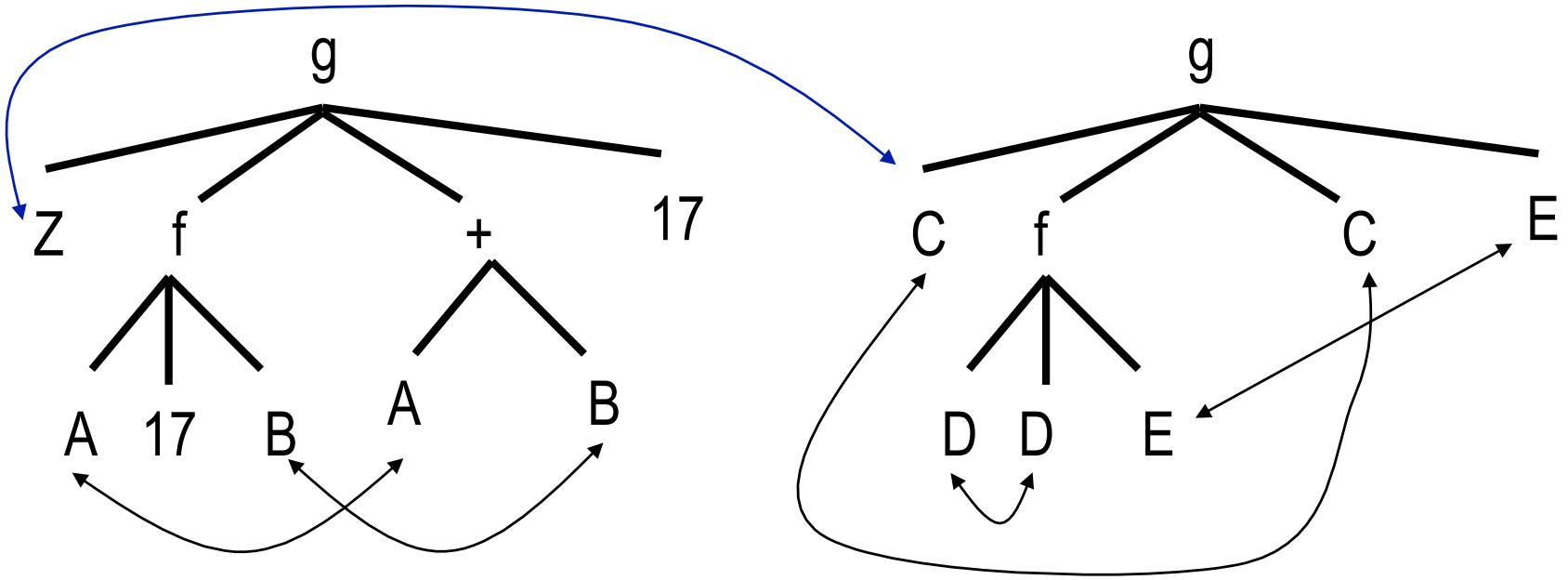
First write in the co-referring variables.



Exercise

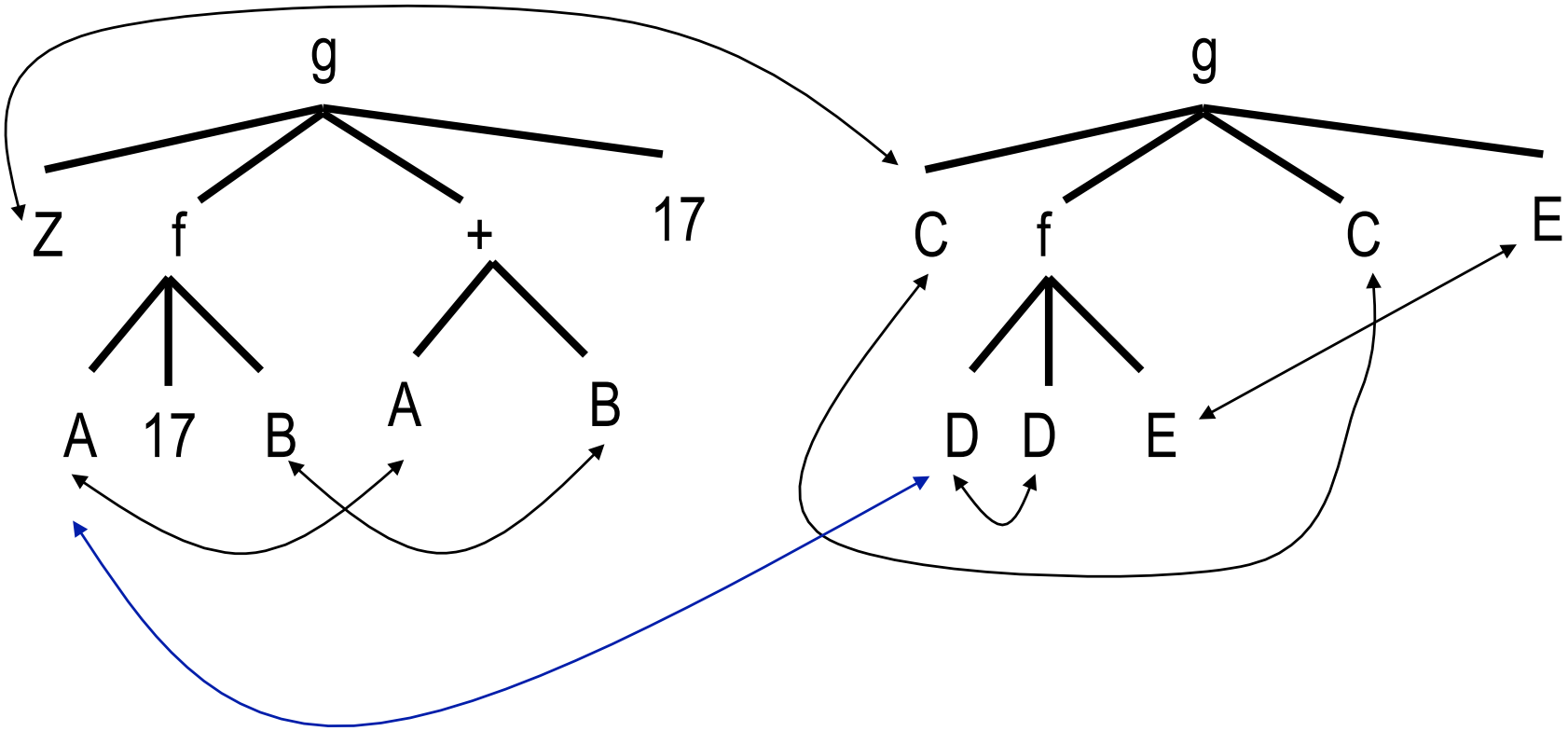
Z/C, C/Z

Now proceed by recursive descent
We go top-down, left-to-right, but the order does not matter as long as it is systematic and complete.



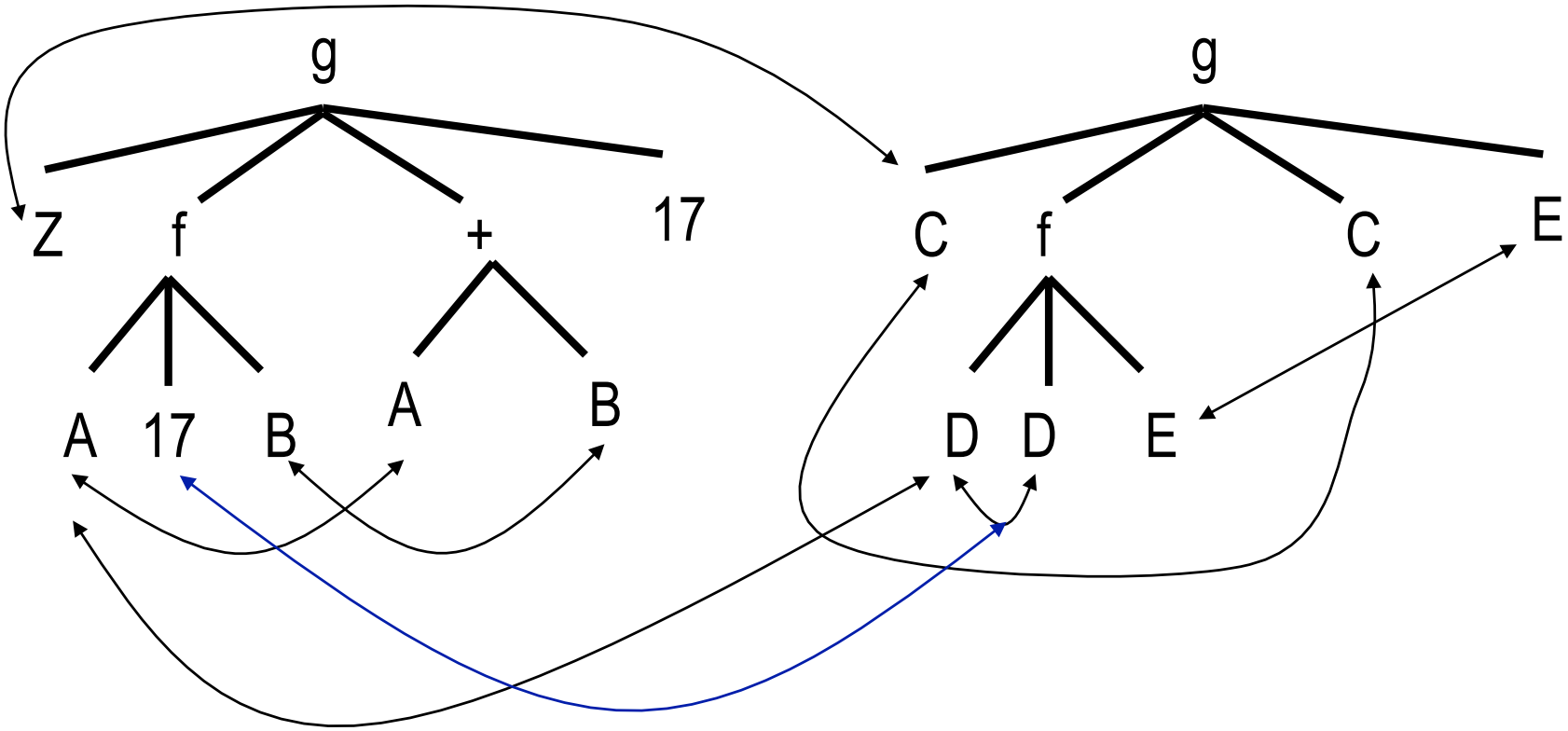
Exercise

Z/C, C/Z, A/D, D/A



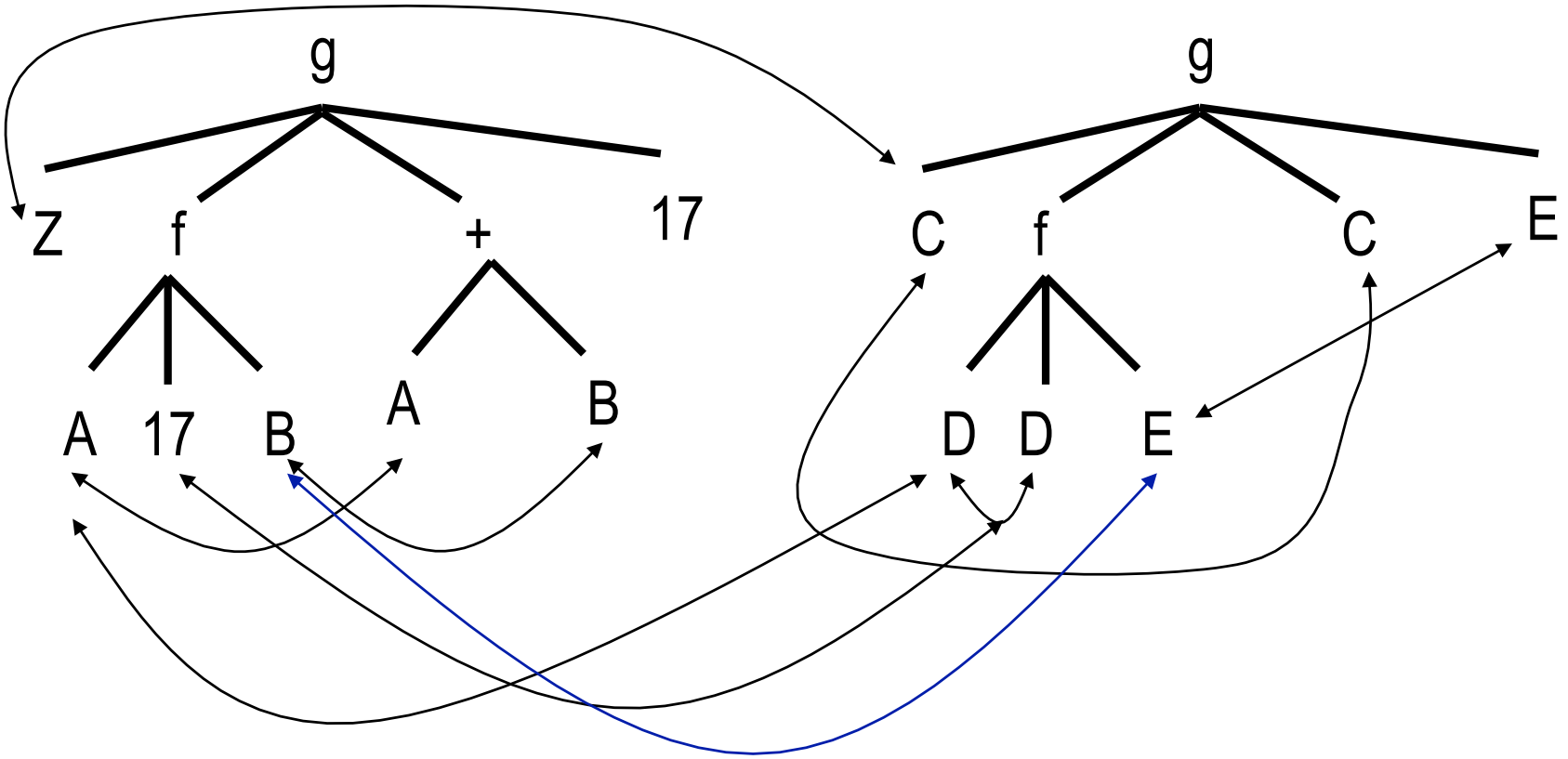
Exercise

Z/C, C/Z, A/17, D/17



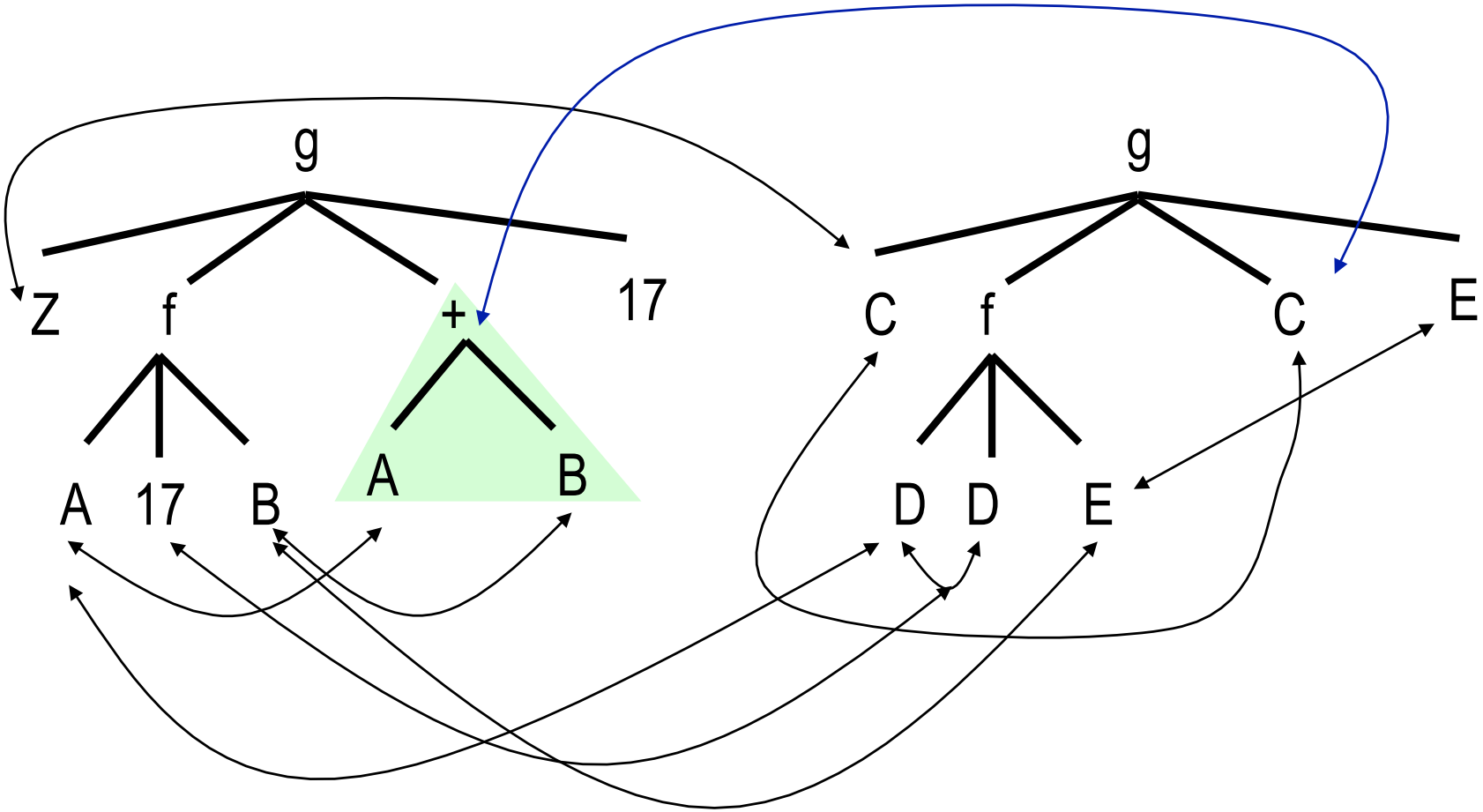
Exercise

Z/C, C/Z, A/17, D/17, B/E, E/B



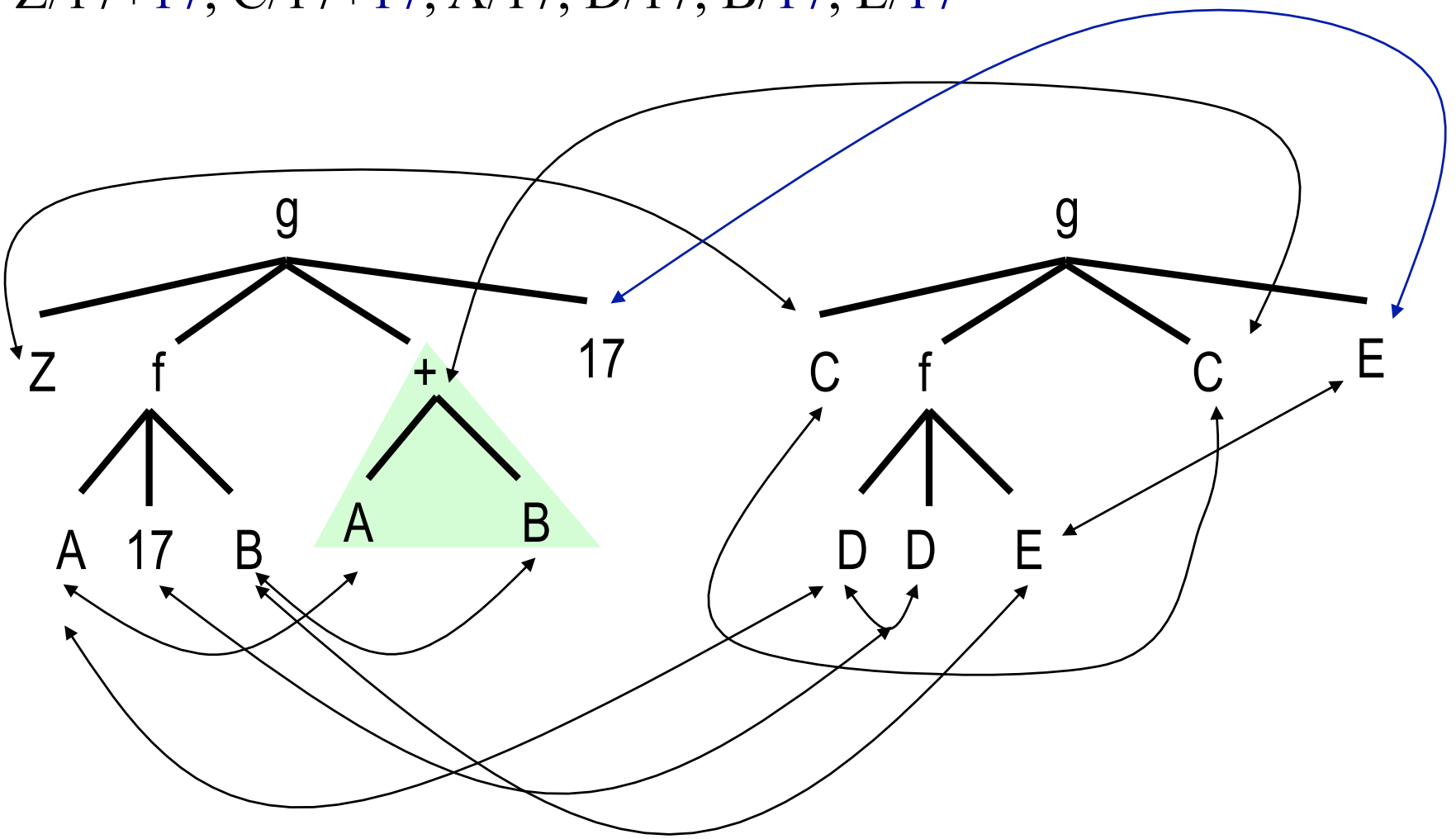
Exercise

Z/17+B, C/17+B, A/17, D/17, B/E, E/B



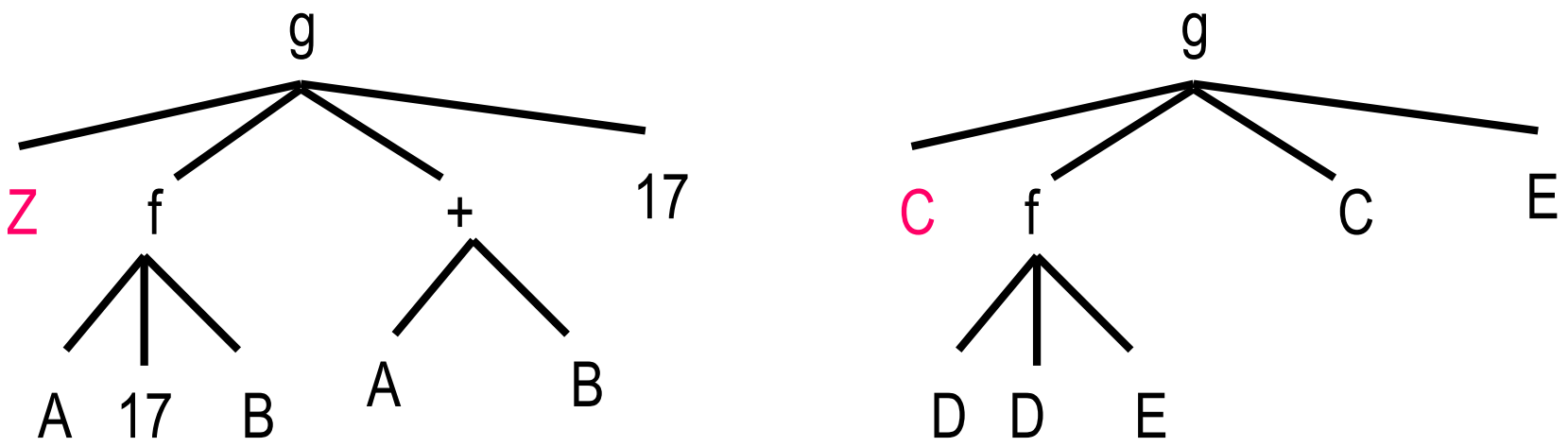
Exercise

Z/17+17, C/17+17, A/17, D/17, B/17, E/17



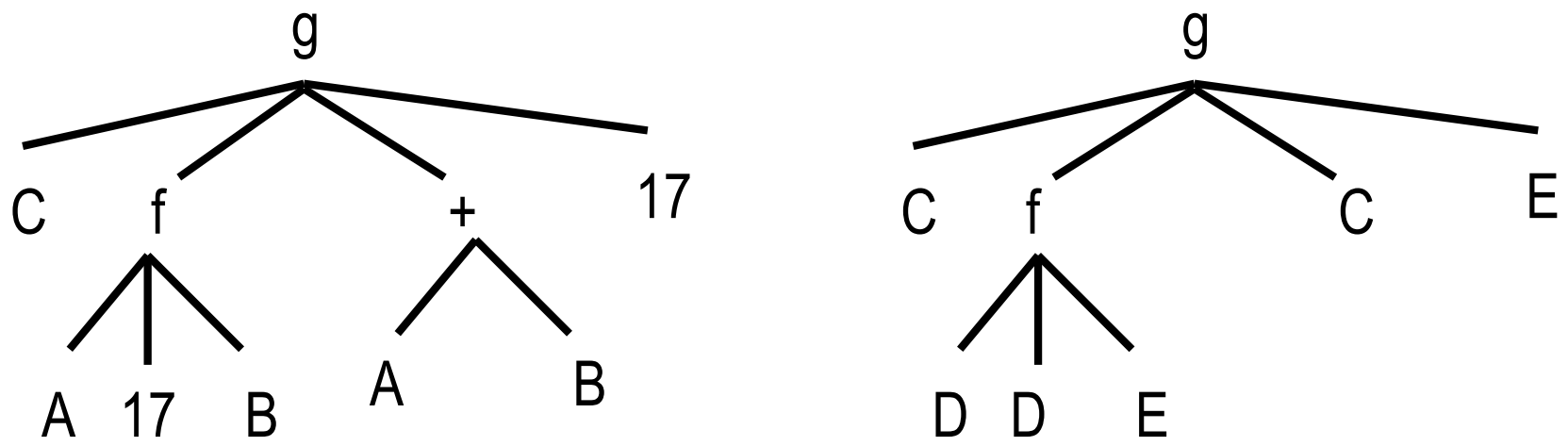
Exercise – Alternative Method

Z/C



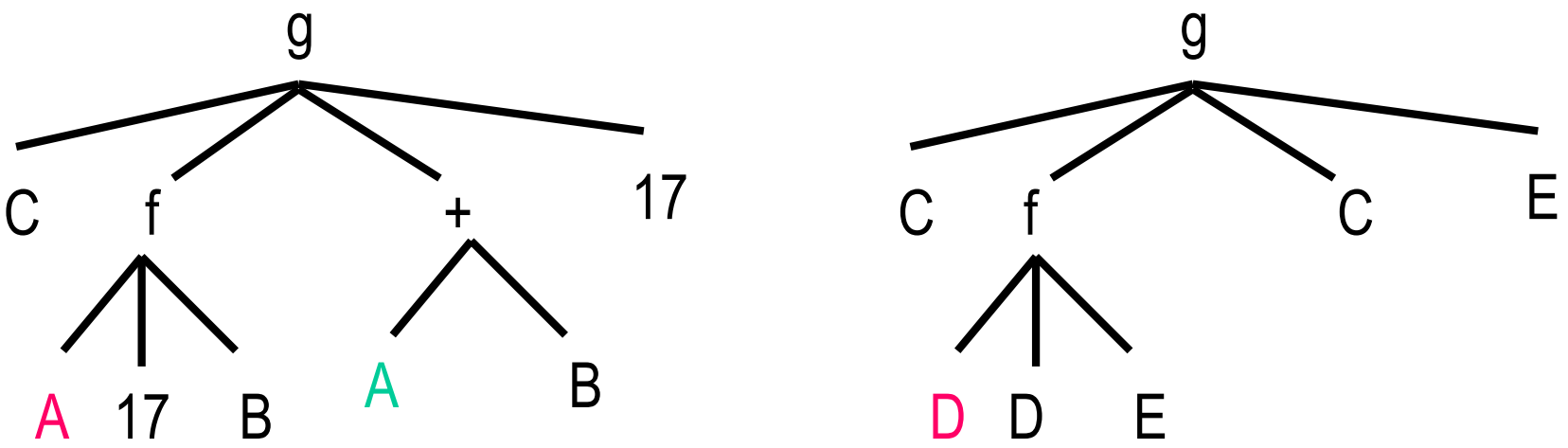
Exercise – Alternative Method

Z/C



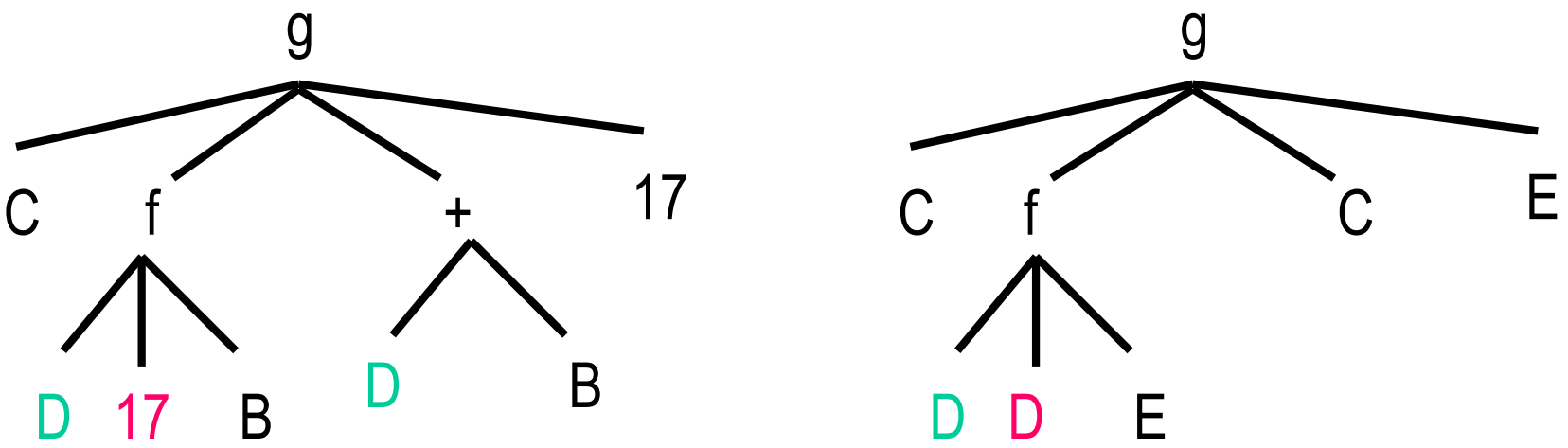
Exercise – Alternative Method

A/D, Z/C



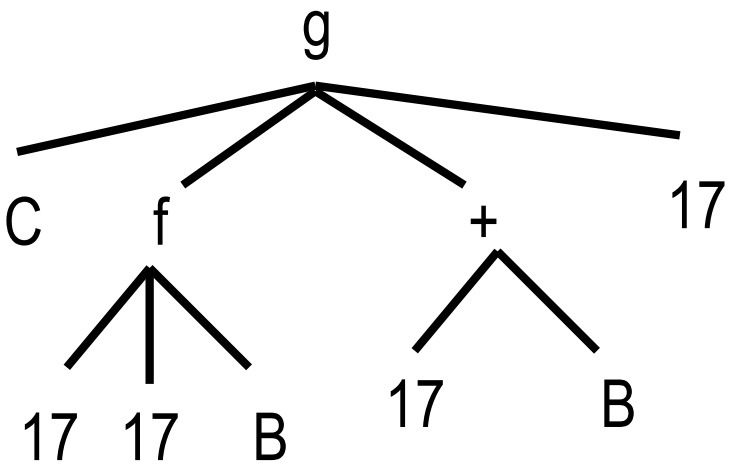
Exercise – Alternative Method

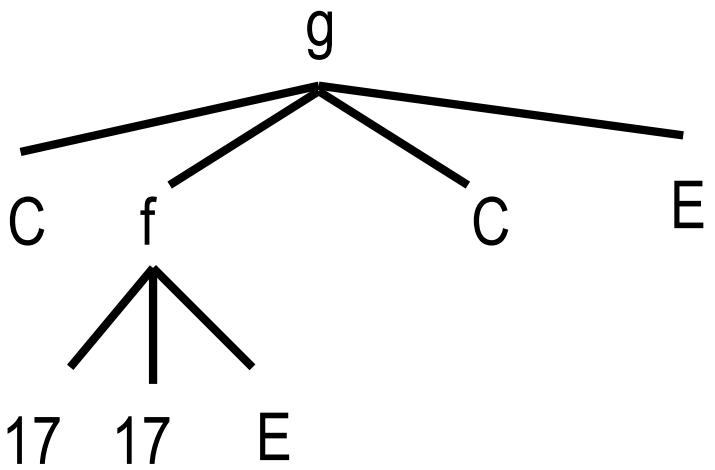
D/17, A/D, Z/C



Exercise – Alternative Method

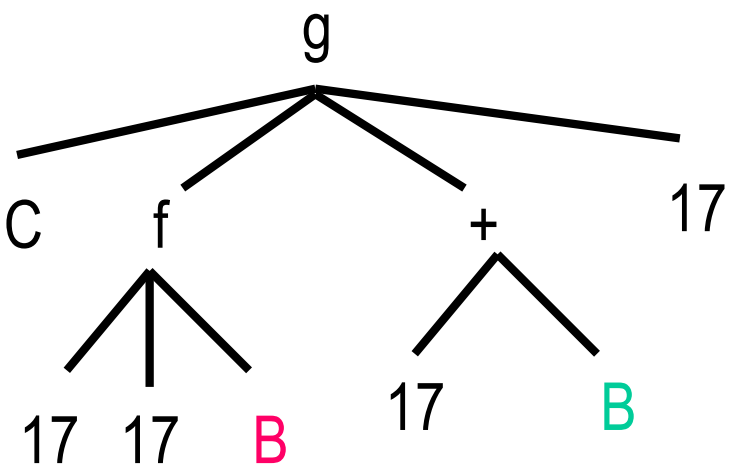
D/17, A/17, Z/C

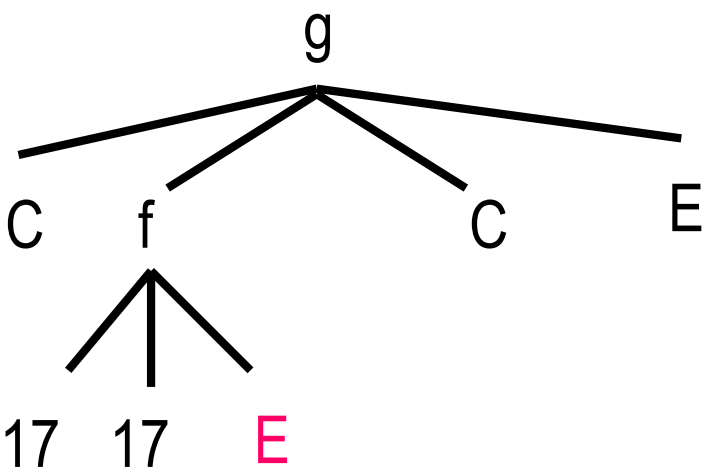




Exercise – Alternative Method

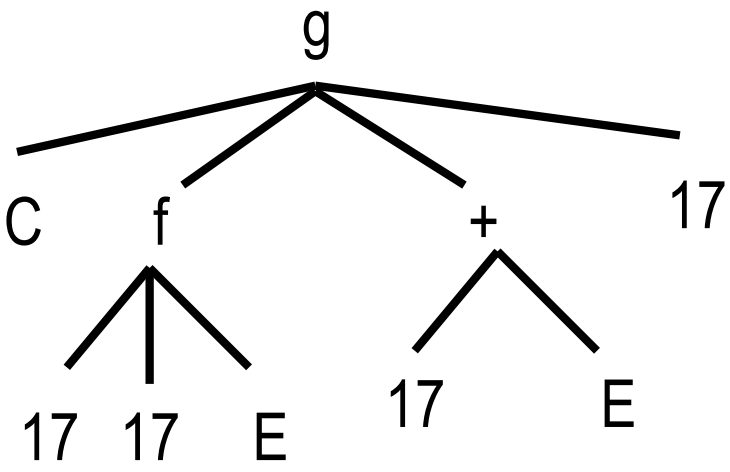
B/E, D/17, A/17, Z/C

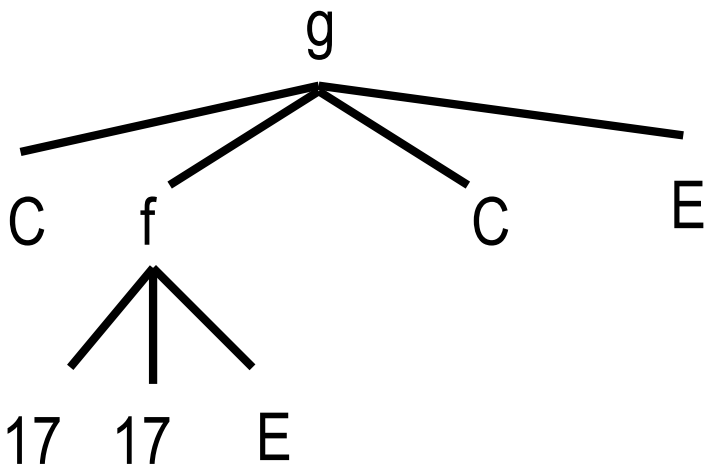




Exercise – Alternative Method

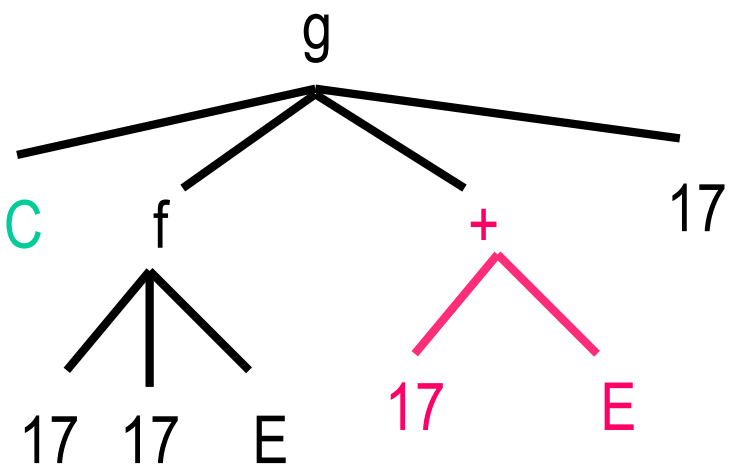
B/E, D/17, A/17, Z/C

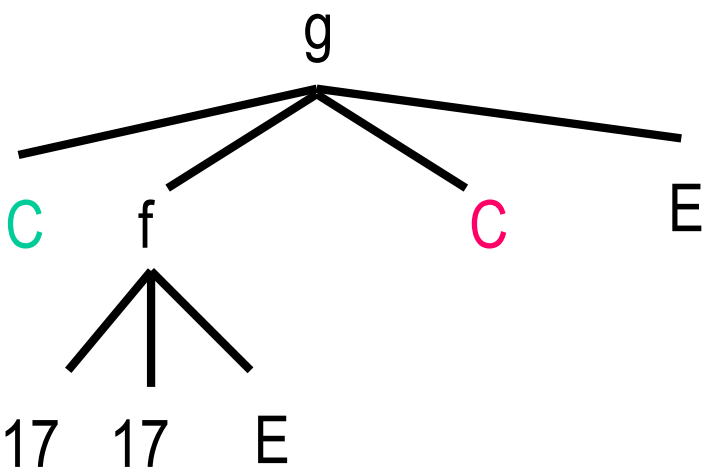




Exercise – Alternative Method

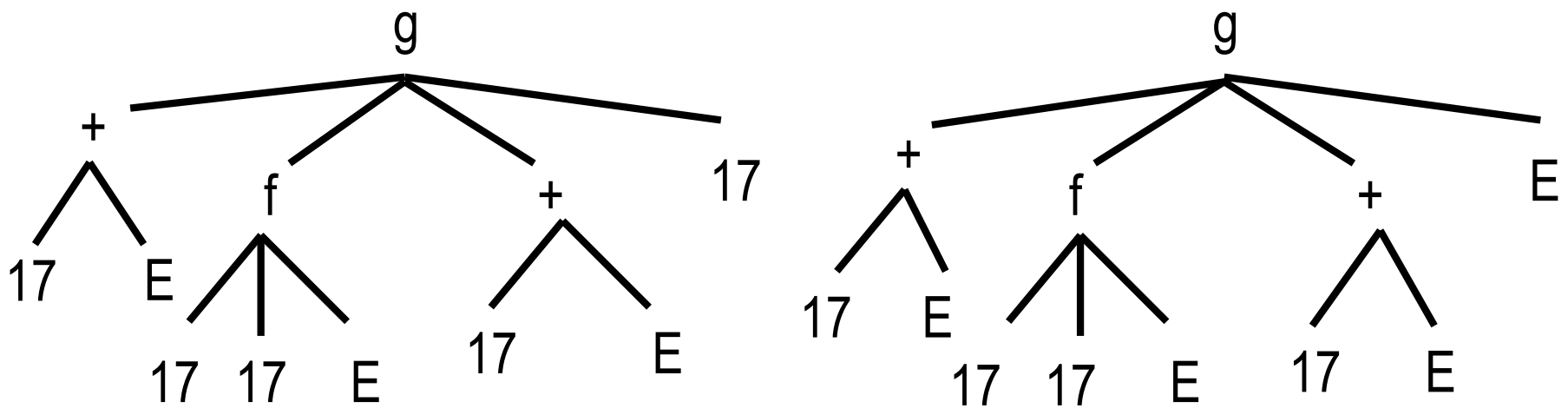
C/17+E, B/E, D/17, A/17, Z/C





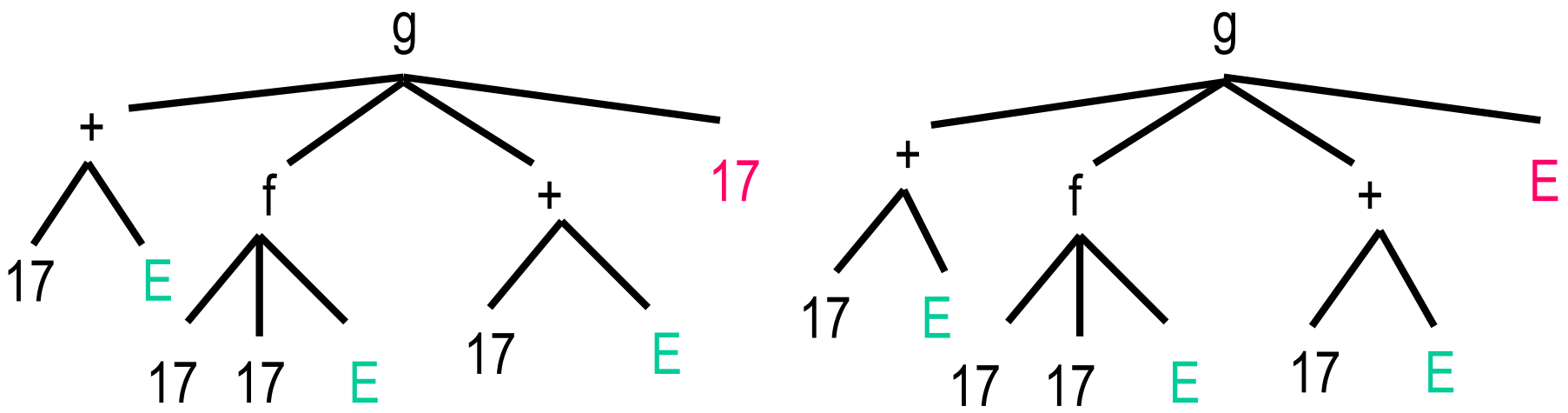
Exercise – Alternative Method

C/17+E, B/E, D/17, A/17, Z/17+E



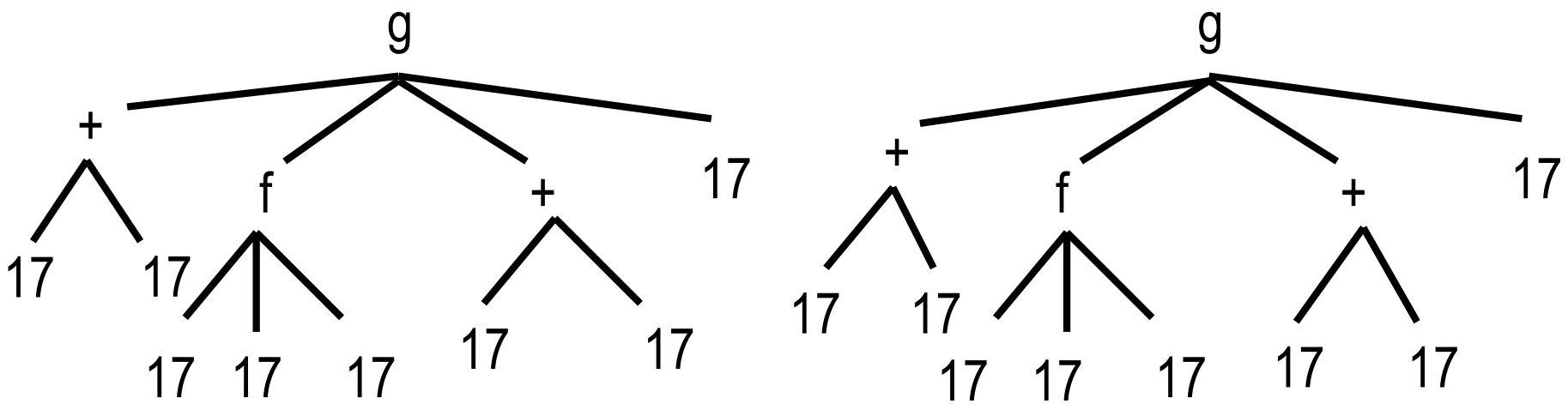
Exercise – Alternative Method

E/17, C/17+E, B/E, D/17, A/17, Z/C



Exercise – Alternative Method

E/17, C/17+17, B/17, D/17, A/17, Z/C



Lists

- Lists are the same as other languages (such as ML) in that a list of terms of any length is composed of list cells that are ‘consed’ together.
- The list of length 0 is called nil, written [].
- The list of length *n* is .(*head*,*tail*), where *tail* is a list of length *n*–1.
- So a list cell is a functor ‘.’ of arity 2. Its first component is the head, and the second component is the tail.

Examples of lists

nil

.(a, nil)

.(a, .(b, nil)

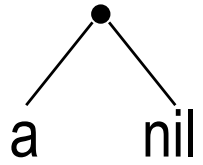
.(a, .(b, .(c, .(d, .(e. nil)))))

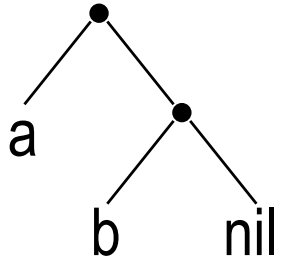
.(a,b) *(note this is a pair, not a proper list)*

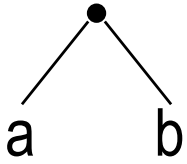
.(a, X) *(this might be a list, or might not!)*

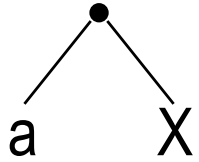
.(a, .(b, nil)), .(c, nil))

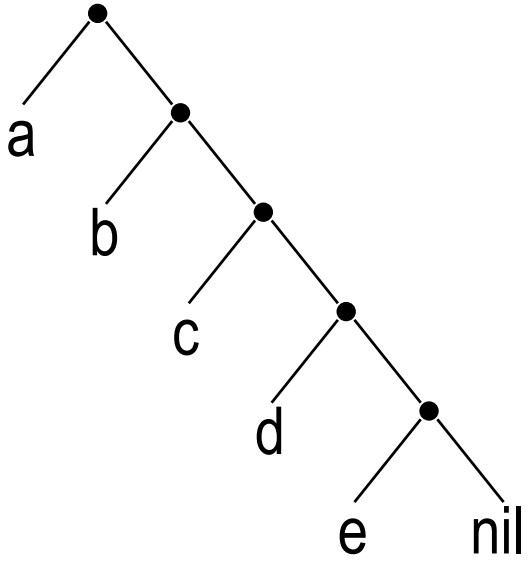
They can be written as trees

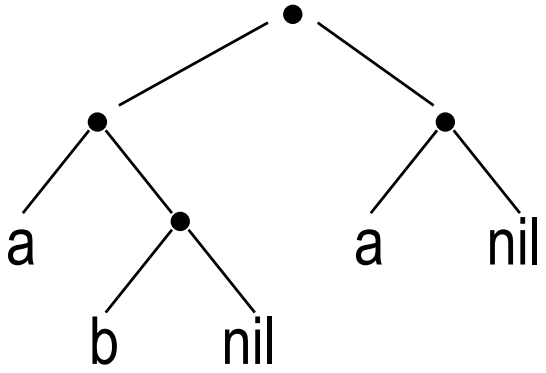












Prolog Syntax for Lists

Nil is written [].

The list consisting of n elements $t_1, t_2, \ldots, t_n$ is written $[t_1, t_2, \ldots, t_n]$.

.(X,Y) is written [X|Y]

[X|[]] is written [X]

The term .(a, .(b, .(c,Y))) is written [a,b,c|Y].

If Y is instantiated to [], then the term is a list, and can be written [a,b,c|[]] or simply [a,b,c].

Exercises

Identify the heads and tails of these lists (if any):

[a, b, c]

[a]

[]

[[the, cat], sat]

[[the, cardinal], [pulled, [off]], [each, [plum, coloured], shoe]

Exercises

Identify the heads and tails of these lists (if any):

a [b, c]

[a]

[]

[[the, cat], sat]

[[the, cardinal], [pulled, [off]], [each, [plum, coloured], shoe]

Exercises

Identify the heads and tails of these lists (if any):

[a, b, c]

a []

[]

[[the, cat], sat]

[[the, cardinal], [pulled, [off]], [each, [plum, coloured], shoe]

Exercises

Identify the heads and tails of these lists (if any):

[a, b, c]

[a]

[] *(not a list, so doesn't have head and tail. nil is a constant)*

[[the, cat], sat]

[[the, cardinal], [pulled, [off]], [each, [plum, coloured], shoe]

Exercises

Identify the heads and tails of these lists (if any):

[a, b, c]

[a]

[]

[the, cat] [sat]

[[the, cardinal], [pulled, [off]], [each, [plum, coloured], shoe]

Exercises

Identify the heads and tails of these lists (if any):

[a, b, c]

[a]

[]

[[the, cat], sat]

[the, cardinal] [pulled, [off]], [each, [plum, coloured], shoe]

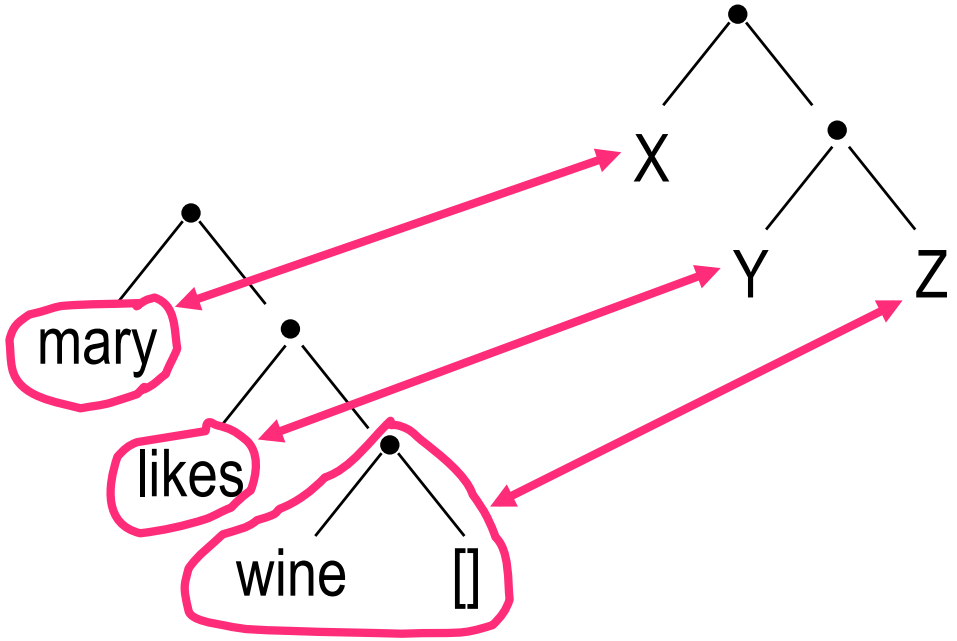
Exercises

For each pair of terms, determine whether they unify, and if so, to which terms are the variables instantiated?

[X, Y, Z]	[john, likes, fish]
[cat]	[X|Y]
[X,Y|Z]	[mary, likes, wine] *(picture on next slide)*
[[the,Y]|Z]	[[X,answer], [is, here]]
[X, Y, X]	[a, Z, Z]
[[X], [Y], [X]]	[[a], [X], [X]]

Remember

A variable may be instantiated to any term.



[mary, likes, wine]

[X,Y|Z]

Fun with Lists (Worksheet 5)

```
/* member(Term, List) */
member(X, [X|T]).
member(X, [H|T]) :- member(X, T).
```

Examples:

```
?- member(john, [paul, john]).
?- member(X, [paul, john]).
?- member(joe, [marx, darwin, freud]).
?- member(foo, X).
```

Exercises

Here is a mystery predicate. What does it do?

```
mystery(X, A, B) :- member(X, A), member(X, B).
```

```
?- mystery(a, [b, c, a], [p, a, l]).
?- mystery(b, [b, l, u, e], [y, e, l, l, o, w]).
?- mystery(X, [r, a, p, i, d], [a, c, t, i, o, n]).
?- mystery(X, [w, a, l, n, u, t], [c, h, e, r, r, y]).
```

A Brief Diversion into Anonymous Variables

```
/* member(Term, List) */

member(X, [X|T]).

member(X, [H|T]) :- member(X, T).

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Notice T isn't 'used'

Notice H isn't 'used'

A Brief Diversion into Arithmetic

The built-in predicate ‘is’ takes two arguments. It interprets its second as an arithmetic expression, and unifies it with the first. Also, ‘is’ is an infix operator.

```
?- X is 2 + 2 * 2.
X = 6
?- 10 is (2 * 0) + 2 << 4.
no
?- 32 is (2 * 0) + 2 << 4.
yes
```

is

But ‘is’ cannot solve equations, so the expression must be ‘ground’ (contain no free variables.

?- Y is 2 * X.

no

?- X is 15, Y is 2 * X.

X = 15, Y= 30.

Worksheet 6: Length of a List

/* length(List, Length) */

Naïve method:

```
length([], 0).
length([H|T], N) :- length(T, NT), N is NT + 1.
```

Worksheet 6

```
/* length(List, Length) */
```

Tail-recursive method:

```
length(L, N) :- acc(L, 0, N).

/* acc(List, Before, After) */

acc([], A, A).
acc([H|T], A, N) :- A1 is A + 1, acc(T, A1, N).
```

Exercises

?- length([apple, pear], N).

?- length([alpha], 2).

?- length(L, 3).

Modify length to give a procedure sum such that sum(L,N) succeeds if L is a list of integers and N is their sum.

Worksheet 7: Inner Product

A list of n integers can be used to represent an n-vector (a point in n-dimensional space). Given two vectors a and b, the inner product (dot product) of a and b is defined

$$\underline{a} \cdot \underline{b} = \sum_{i=1}^{n} a_i b_i$$

As you might expect, there are naïve and tail-recursive ways to compute this.

Worksheet 7

The naïve method:

```
inner([], [], 0).
inner([A|As],[B|Bs],N) :-
        inner(As, Bs, Ns), N is Ns + (A * B).
```

Worksheet 7

Tail-recursive method:

```
inner(A, B, N) :- dotaux(A, B, 0, N).

dotaux([], [], V, V).
dotaux([A|As],[B|Bs],N,Z) :-
        N1 is N + (A * B),
        dotaux(As, Bs, N1, Z).
```

Worksheet 8: Maximum of a List

Tail-recursive method has a base case and two recursive cases:

```
/*  max(List, Accumulator, Result) */

max([], A, A).
max([H|T], A, M) :- H > A, max(T, H, M).
max([H|T], A, M) :- H =< A, max(T, A, M).
```

How to initialise the accumulator?

Worksheet 8

maximum(L, M) :- max(L, -10000, M).

Magic numbers are a bad idea.

maximum(L, M) :- max(L, minint, M)

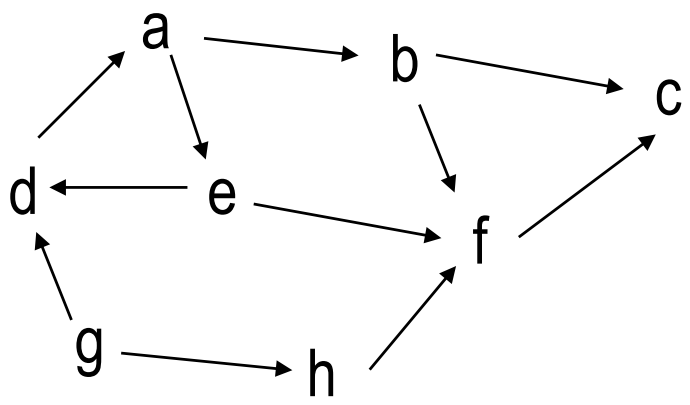
Need to change definitions of arithmetic.

maximum([H|T], M) :- max(T, H, M).

And know that ?- maximum([],X) *fails.*

Worksheet 9


```
a(g, h).
a(d, a).
a(g, d).
a(e, d).
a(h, f).
a(e, f).
a(a, e).
a(a, b).
a(b, f).
a(b, c).
a(f, c).
```



Here we have added a new arc from d to a.
If you use the program from WS 4, some goals will cause an infinite loop.

Keep a 'trail' of nodes visited so far. Visit only 'legal' nodes, not already on the trail. Represent the trail as an accumulator, an extra argument of path.

Worksheet 9

```
path(X, X, T).
path(X, Y, T) :-
        a(X, Z), legal(Z, T), path(Z, Y, [Z|T]).

legal(Z, []).
legal(Z, [H|T]) :- Z \== H, legal(Z, T).
```

Notice that legal is like the negation of member.