Mapping

Map

a given structure
to another structure
given a set of rules

traverse the old structure
component by component
construct the new structure
with transformed components

Example

you are a computer

maps to a reply

`i am not a computer`

or

do you speak french

maps to a reply

`no i speak german`

Procedure

Accept a sentence

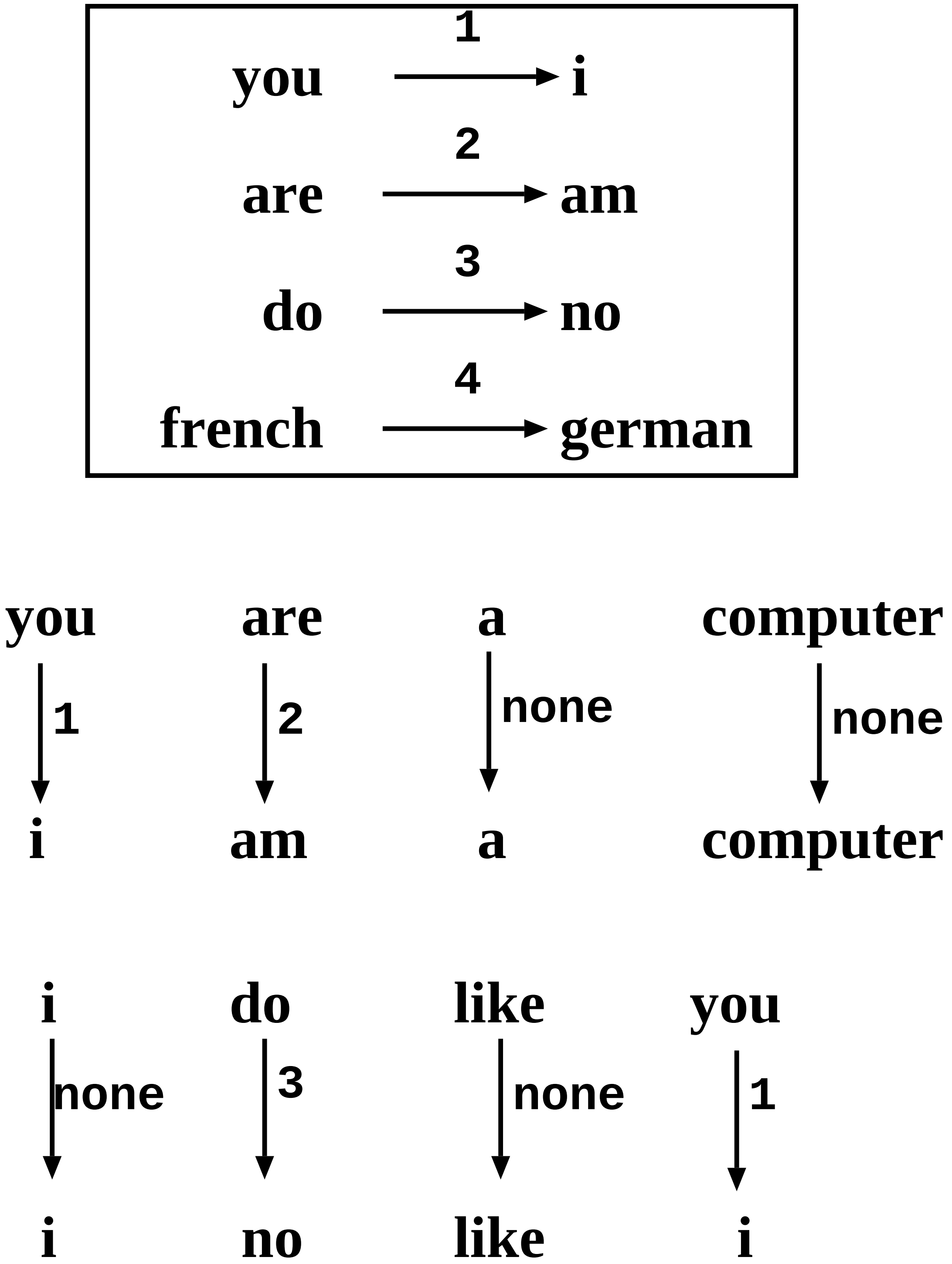
Change **you** to `i`

Change **are** to `am not`

Change **french** to `german`

Change **do** to `no`

Process


you → i (1)
are → am (2)
do → no (3)
french → german (4)
you are a computer
1 2 none none
i am a computer
i do like you
none 3 none 1
i no like i

PROLOG

```
| ?- alter([do,you,know,french],X).

X = [no,i,know,german] ?

yes



change(you,i).
change(are,[am,not]).
change(french,german).
change(do,no).
change(X,X).

alter([],[]).
alter([H|T],[X|Y]) :-
     change(H,X),
     alter(T,Y).
```

Recursion

```
alter([H|T],[X|Y])  :-
      change(H,X),
      alter(T,Y).

alter([],[]).
```

change the head of the word into another word

let that be the output head

alter the tail of the input list

let that be the tail of the output

If we have reached the end

there is nothing more to do

Trace

```
— ?- alter([do,you,know,french],X).
 + 1  1  Call: alter([do,you,know,french],_89) ?
 + 2  2  Call: change(do,_364) ?
 + 2  2  Exit: change(do,no) ?
 + 3  2  Call: alter([you,know,french],_365) ?
 + 4  3  Call: change(you,_919) ?
 + 4  3  Exit: change(you,i) ?
 + 5  3  Call: alter([know,french],_920) ?
 + 6  4  Call: change(know,_1473) ?
 + 6  4  Exit: change(know,know) ?
 + 7  4  Call: alter([french],_1474) ?
 + 8  5  Call: change(french,_2026) ?
 + 8  5  Exit: change(french,german) ?
 + 9  5  Call: alter([],_2027) ?
 + 9  5  Exit: alter([],[]) ?
 + 7  4  Exit: alter([french],[german]) ?
 + 5  3  Exit: alter([know,french],[know,german]) ?
 + 3  2  Exit: alter([you,know,french],[i,know,german]) ?
 + 1  1  Exit: alter([do,you,know,french],
                     [no,i,know,german]) ?
X = [no,i,know,german] ?
```

```
| ?- alter([you,are,a,computer],X).
 + 1  1  Call: alter([you,are,a,computer],
                      _89) ?
 + 2  2  Call: change(you,_364) ?
 + 2  2  Exit: change(you,i) ?

 + 3  2  Call: alter([are,a,computer],
                         _365) ?
 + 4  3  Call: change(are,_919) ?
 + 4  3  Exit: change(are,[am,not]) ?
                      .
                      .
                      .
 + 1  1  Exit: alter([you,are,a,computer],
               [i,[am,not],a,computer]) ?

X = [i,[am,not],a,computer] ?
```

Boundary Conditions

Termination

```
alter([],[]).
```

Catch All

```
change(X,X).
```

If none of the other conditions were satisfied,
(it is not to be changed)
then just return the same.

Recursive Comparison

Comparing Structures

More complicated than the simple integers

Have to compare all the individual components

Break down components recursively

aless

`aless(X,Y)`

Will succeed
if X and Y stand for atoms
and
X is alphabetically less than Y

Succeed

```
aless(avocado,clergyman).
```

Fail

```
aless(windmill,motorcar).
aless(picture,picture).
```

Success

Success

First word ends before second

`aless(book,bookbinder).`

Success

A character in the first
is alphabetically less
than one in the second

`aless(avocado,clergyman).`

Recursion

The first character is the same in both
Then have to check the rest

`aless(lazy,leather).`

check

`aless(azy,eather).`

Failure

Fail

Reach the end of both words
at the same time

aless(apple,apple).

Fail

Run out of characters for the second word

```
aless(alphabetic,alp)
```

Representation

A list of ASCII codes

Atoms to List

Intrinsic Function

`name(AtomName,List).`

```
| ?- name(alp,[97,108,112]).
yes
| ?- name(alp,X).
X = [97,108,112] ?
yes
| ?- name(X,[97,108,112]).
X = alp ?
yes
```

First Task

Need to convert atom to list

and

then compare with this list

Need

`name(X,XList)`

and `name(Y,YList)`

Need to compare lists

`alessx(XLIst,YList).`

Put it together

```
aless(X,Y) :-
name(X,XList),
name(Y,YList),
alessx(XLIst,YList).
```

alessx Conditions

Success

First word ends before second

(First word is empty and the second is not)

```
alessx([],[_,_]).
```

Success

The first character in the first

is alphabetically less

than the first character in the second

```
alessx([X|_],[Y|_]) :- X< Y.
```

alessx Conditions

Recursion

The first character is the same in both

Then have to check the rest

```
aless([A|X],[B|Y]) :-
A = B, alessx(X,Y).
```

Equivalently

```
aless([H|X],[H|Y]) :-
alessx(X,Y).
```

Failure

Fail

Reach the end of both words
at the same time

aless(apple,apple).

Fail

Run out of characters for the second word

`aless(alphabetic,alp)`

abc < bcd

```
| ?- aless(abc,bcd).
 + 1  1  Call: aless(abc,bcd) ?
 + 2  2  Call: name(abc,_322) ?
 + 2  2  Exit: name(abc,[97,98,99]) ?
 + 3  2  Call: name(bcd,_316) ?
 + 3  2  Exit: name(bcd,[98,99,100]) ?
 + 4  2  Call: alessx([97,98,99],
                      [98,99,100]) ?
 + 5  3  Call: 97<98 ?
 + 5  3  Exit: 97<98 ?
 + 4  2  Exit: alessx([97,98,99],
                      [98,99,100]) ?
 + 1  1  Exit: aless(abc,bcd) ?
```

bcd < abc

```
| ?- aless(bcd,abc).
 + 1  1  Call: aless(bcd,abc) ?
 + 2  2  Call: name(bcd,_322) ?
 + 2  2  Exit: name(bcd,[98,99,100]) ?
 + 3  2  Call: name(abc,_316) ?
 + 3  2  Exit: name(abc,[97,98,99]) ?
 + 4  2  Call: alessx([98,99,100],
                      [97,98,99]) ?
 + 5  3  Call: 98<97 ?
 + 5  3  Fail: 98<97 ?
 + 4  2  Fail: alessx([98,99,100],
                      [97,98,99]) ?
 + 3  2  Redo: name(abc,[97,98,99]) ?
 + 3  2  Fail: name(abc,_316) ?
 + 2  2  Redo: name(bcd,[98,99,100]) ?
 + 2  2  Fail: name(bcd,_322) ?
 + 1  1  Fail: aless(bcd,abc) ?
```

Test

```
+ 4  2  Call: alessx([97,98,99,99],
                     [97,98,99,100]) ?
+ 5  3  Call: 97<97 ?
+ 5  3  Fail: 97<97 ?
+ 5  3  Call: alessx([98,99,99],
                     [98,99,100]) ?
+ 6  4  Call: 98<98 ?
+ 6  4  Fail: 98<98 ?
+ 6  4  Call: alessx([99,99],[99,100]) ?
+ 7  5  Call: 99<99 ?
+ 7  5  Fail: 99<99 ?
+ 7  5  Call: alessx([99],[100]) ?
+ 8  6  Call: 99<100 ?
+ 8  6  Exit: 99<100 ?
+ 7  5  Exit: alessx([99],[100]) ?
+ 6  4  Exit: alessx([99,99],[99,100]) ?
+ 5  3  Exit: alessx([98,99,99],
                     [98,99,100]) ?
+ 4  2  Exit: alessx([97,98,99,99],
                     [97,98,99,100]) ?
```

Test

```
+ 4  2  Call: alessx([97,98,99],
                      [97,98,99]) ?
+ 5  3  Call: 97<97 ?
+ 5  3  Fail: 97<97 ?
+ 5  3  Call: alessx([98,99],[98,99]) ?
+ 6  4  Call: 98<98 ?
+ 6  4  Fail: 98<98 ?
+ 6  4  Call: alessx([99],[99]) ?
+ 7  5  Call: 99<99 ?
+ 7  5  Fail: 99<99 ?
+ 7  5  Call: alessx([],[]) ?
+ 7  5  Fail: alessx([],[]) ?
+ 6  4  Fail: alessx([99],[99]) ?
+ 5  3  Fail: alessx([98,99],[98,99]) ?
+ 4  2  Fail: alessx([97,98,99],
                      [97,98,99]) ?
```

Joining Structures

Joining two lists together

True or False

```
append([a,b,c],[3,2,1],[a,b,c,3,2,1]).
```

What is the Total List

```
append([alpha,beta],[gamma,delta],X).
X = [alpha,beta,gamma,delta]
```

Isolate the first joined part

```
append(X,[b,c,d],[a,b,c,d]).
X = [a] ?
```

Program

```
append([],L,L).

append([X|L1],L2,[X|L3]) :-
    append(L1,L2,L3).
```

The first element of the first is the first element of the third

The tail of the first list (L1) will always have the second argument (L2) appended to it to form the tail of the third argument (L3)

Recursively use append on the rest of the list

Will be reduced to the empty list (the boundary condition).

Inventory Example

Bicycle Factory

Inventory of Bicycle Parts

Hierarchial Structure

Bicycle made up of parts

Each part made up of sub-parts

Basic Parts

```
basicpart(rim).
basicpart(spoke).
basicpart(rearframe).
basicpart(handles).
basicpart(gears).
basicpart(bolt).
basicpart(nut).
basicpart(fork).
```

Assemble Parts

```
assembly(bike,[wheel,wheel,frame]).
assembly(wheel,[spoke,rim,hub]).
assembly(frame,[rearframe,frontframe]).
assembly(frontframe,[fork,handles]).
assembly(hub,[gears,axle]).
assembly(axle,[bolt,nut]).
```