



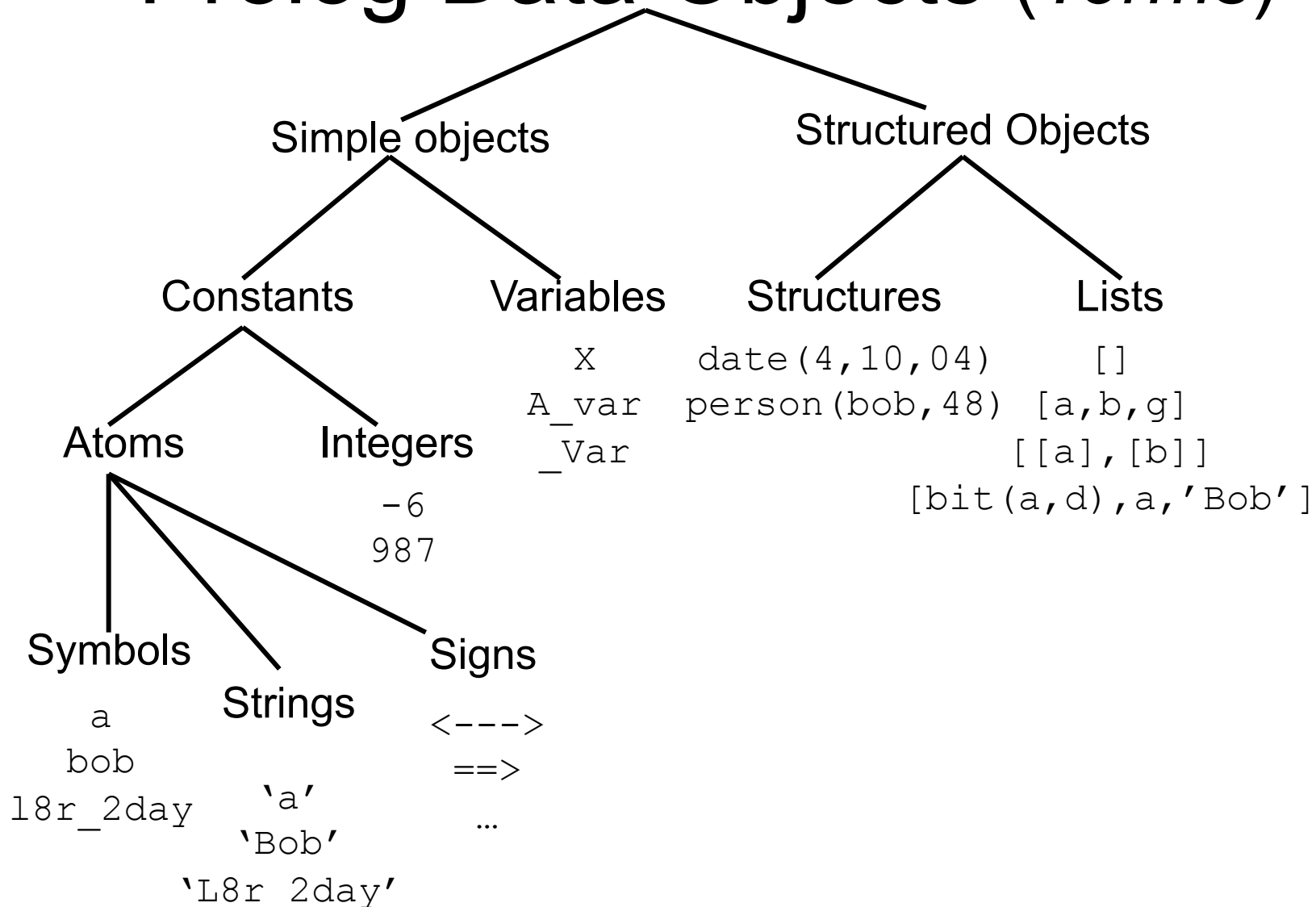
Recursion, Structures, and Lists

Artificial Intelligence Programming in Prolog
Lecture 4

Why use recursion?

- It allows us to define very **clear** and **elegant code**.
 - Why repeat code when you can reuse existing code.
- Relationships may be recursive
 - e.g. “X is my ancestor if X is my Ancestor’s ancestor.”
- Data is represented recursively and best processed iteratively.
 - Grammatical structures can contain themselves
 - E.g. $NP \rightarrow (Det) N (PP)$, $PP \rightarrow P (NP)$
 - Ordered data: each element requires the same processing
- Allows Prolog to perform complex search of a problem space without any dedicated algorithms.

Prolog Data Objects (*Terms*)



Structures

- To create a single data element from a collection of related terms we use a *structure*.
- A structure is constructed from a *functor* (a constant symbol) and one or more *components*.

somerelationship(*a*,*b*,*c*,*[1,2,3]*)

- The components can be of any type: atoms, integers, variables, or structures.
- As functors are treated as data objects just like constants they can be unified with variables

```
|?- X = date(04,10,04) .  
X = date(04,10,04) ?  
yes
```

Structure unification

- 2 structures will unify if
 - the functors are the same,
 - they have the same number of components,
 - and all the components unify.

```
| ?- person(Nm,london,Age) = person(bob,london,48) .
```

```
Nm = bob,
```

```
Age = 48?
```

```
yes
```

```
| ?- person(Someone,_,45) = person(harry,dundee,45) .
```

```
Someone = harry ?
```

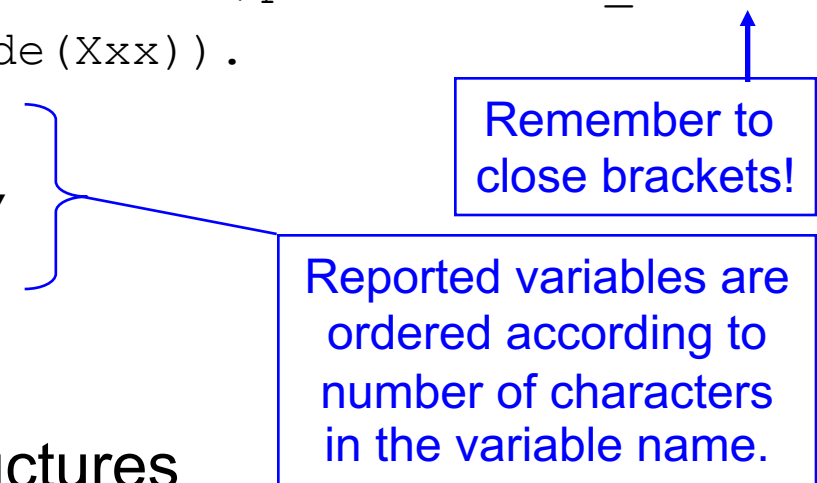
```
yes
```

- (A plain underscore '_' is not bound to any value. By using it you are telling Prolog to ignore this argument and not report it.)

Structure unification (2)

- A structure may also have another structure as a component.

```
|?-addr(flat(4),street('Home Str.'),postcode(eh8_9lw))  
= addr(flat(Z),Yy,postcode(Xxx)).  
Z = 4,  
Yy = street('Home Str.'),  
Xxx = eh8_9lw ?  
yes
```



Remember to close brackets!

Reported variables are ordered according to number of characters in the variable name.

- Unification of nested structures works **recursively**:
 - first it unifies the entire structure,
 - then it tries to unify the nested structures.

Structures = facts?

- The syntax of structures and facts is identical but:
 - Structures are not facts as they are not stored in the database as being true (followed by a period '.');
 - Structures are generally just used to group data;
 - Functors do not have to match predicate names.
- However predicates can be stored as structures

command(**X**) :-

X.

By instantiating a variable with a structure which is also a predicate you can pass commands.

```
| ?- X = write('Passing a command'), command(X) .
```

```
    Passing a command
```

```
X = write('Passing a command') ?
```

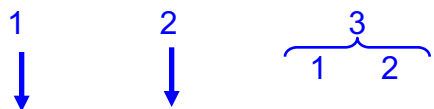
```
yes
```

Lists

- A collection of ordered data.
- Has **zero** or more elements enclosed by **square brackets** ('[]') and **separated by commas** (',').

[a] ← a list with one element

[] ← an empty list


[34, tom, [2, 3]] ← a list with 3 elements where the
3rd element is a list of 2 elements.

- Like any object, a list can be unified with a variable

```
|?- [Any, list, 'of elements'] = X.
```

```
X = [Any, list, 'of elements']?
```

```
yes
```


List Unification

- Two lists unify if they are the same length and all their elements unify.

$| ? - [a, B, c, D] = [A, b, C, d] .$ $| ? - [(a+X), (Y+b)] = [(W+c), (d+b)] .$

$A = a,$

$W = a,$

$B = b,$

$X = c,$

$C = c,$

$Y = d?$

$D = d ?$

yes

yes

$| ? - [[X, a]] = [b, Y] .$

$| ? - [[a], [B, c], []] = [X, [b, c], Y] .$

no

$B = b,$

$X = [a],$

$Y = [] ?$

yes



Definition of a List

- Lists are *recursively defined* structures.

“An empty list, [], is a list.

A structure of the form [X, ...] is a list if X is a term and [...] is a list, possibly empty.”

- This recursiveness is made explicit by the bar notation
 - [Head|Tail] (‘|’ = bottom left PC keyboard character)
- Head must unify with a single term.
- Tail unifies with a list of any length, including an empty list, [].
 - the bar notation turns everything after the Head into a list and unifies it with Tail.

Head and Tail

`|?-[a,b,c,d]=[Head|Tail].`

`Head = a,`

`Tail = [b,c,d]?`

`yes`

`|?-[a] = [H|T].`

`H = a,`

`T = [];`

`yes`

`|?-[] = [H|T].`

`no`

`|?-[a,b,c,d]=[X|[Y|Z]].`

`X = a,`

`Y = b,`

`Z = [c,d];`

`yes`

`|?-[a,b,c]=[W|[X|[Y|Z]]].`

`W = a,`

`X = b,`

`Y = c,`

`Z = []? yes`

`|?-[a|[b|[c|[]]]]= List.`

`List = [a,b,c]?`

`yes`

Summary

- Tree representations allow us trace Prolog's search for multiple matches to a query.
- They also highlight the strengths and weaknesses of recursion (e.g. economical code vs. infinite looping).
- Recursive data structures can be represented as *structures* or *lists*.
- Structures can be unified with variables then used as commands.
- Lists can store ordered data and allow its sequential processing through recursion.