Prolog Lists

Lists Manipulation in Prolog

- A Non-list Example
- Represent data in the form of lists
- Prolog’s List notation
- List manipulation techniques
- Use built-in predicates to manipulate lists
- Append
- Running definitions forwards and backwards

• Define predicates to work through a list element by element from left to

right using recursion.

This lecture describes a flexible type of data object called a list. It shows how to work through a list element by element from left to right using recursion to perform the same or a similar operation on each element, how to manipulate lists using built-in predicates and how to create a list containing all the possible values that would satisfy a specified goal.

تصف هذه المحاضرة نوعًا من هياكل البيانات يُسمى القائمة. يوضح كيفية التعامل مع عناصر القائمة من اليسار إلى اليمين باستخدام التكرار لإجراء نفس العملية أو عملية مشابهة على كل عنصر، وكيفية التعامل مع القوائم باستخدام المسندات المدمجة، وكيفية إنشاء قائمة تحتوي على جميع القيم الممكنة التي تُلبي هدفًا محددًا.

Representing Data as Lists تمثيل البيانات كقوائم

A list is written as a sequence of values, known as list elements, separated by commas and enclosed in square brackets,

e.g. [dog, cat, fish, man].

هذه القائمة تحتوي على 4 عناصر

A list element does not have to be an atom. It can be any Prolog term, including a bound or unbound variable or another list,

so **[X, Y, mypred(a, b, c), [p, q, r], z]** is a valid list.

A list element that is itself a list is known as a sublist. Lists can have any number of elements, including zero. The list with no elements is known as the empty list and is written as [].

For non-empty lists, the first element is known as the **head**. The list remaining after the first element is removed is called the **tail**.

For example, the head of the list [dog, cat, fish, man] is the atom **dog** and the tail is the list [cat, fish, man].
The head of list **[X, Y, mypred(a, b, c), [p, q, r], z]** is the atom **X**. The tail is the list **[Y, mypred(a, b, c), [p, q, r], z].**

Another example list: [X, 2+2, [a,b,c]], X= head, [2+2, [a,b,c]] = tail.

Some **further examples** of lists are:

[john, mary, 10, robert, 20, jane, X, bill]

[[portsmouth, london, glasgow], [portsmouth, london, edinburgh], [glasgow]]

[[john, 28], [mary, 56, teacher], robert, parent(victoria, albert), [a, b, [c, d, e], f], 28]

1. **[john, 28]**: This is a sublist that contains two elements:
   - john: likely a name (atom).
   - 28: a number (atom).
2. **[mary, 56, teacher]**: Another sublist with three elements:
   - mary: another name (atom).
   - 56: a number (atom).

    - teacher: possibly a profession (atom).
3. **robert**: An atom representing a name. It stands alone and isn't part of any sublist.
4. **parent(victoria, albert)**: A compound structure (or term) that likely represents a relationship:
    - parent: this could be a functor or a predicate.
    - (victoria, albert): these are the arguments indicating that victoria is a parent of albert.
5. **[a, b, [c, d, e], f]**: A more complex sublist containing:
    - a and b: atoms.
    - [c, d, e]: another sublist.
    - f: an atom.
      This illustrates nesting, where a sublist is contained within another list.

يوضح هذا التعشيش، حيث يتم تضمين قائمة فرعية داخل قائمة أخرى.

6. **28**: Another standalone atom, which is a number.

تحتوي القائمة على أنواع مختلفة من العناصر: الأسماء، والأرقام، وهيكل العلاقات.
توجد قائمتان فرعيتان ([جون، ٢٨] و[ماري، ٥٦، مُعلمة]، [أ، ب، [ج، د، هـ]، و]).
يُشير مصطلح مُركّب (الأصل (فيكتوريا، ألبرت)) إلى علاقة.
يُمثل هذا المصطلح مجموعة بيانات يُمكن استخدامها لنمذجة أشياء مثل الأشخاص وسماتهم، بالإضافة إلى بعض العلاقات المُحتملة.

Notation for Lists تدوين القوائم

Up to now lists have been written as a sequence of list elements written in order, separated by commas and enclosed in square brackets. We will call this 'standard bracketed notation'.

Lists are generally, although not always, written in this notation in queries entered by the user at the system prompt, for example

حتى الآن، كانت القوائم تُكتب كتسلسل من عناصر القائمة، مُرتبة، مفصولة بفواصل ومُحاطة بأقواس مربعة. سنُطلق على هذا "الترميز القياسي المُحاط بالأقواس". تُكتب القوائم عادةً، وإن لم يكن دائمًا، بهذا الترميز في الاستعلامات التي يُدخلها المستخدم عند مُوجه النظام، على سبيل المثال:

```
?- X=alpha,Y=27,Z=[alpha,beta],write('List is: '),write([X,Y,Z]),nl.

List is: [alpha,27,[alpha,beta]]

X = alpha,

Y = 27,

Z = [alpha, beta].
```

مع ذلك (باستثناء القائمة الفارغة)، نادرًا ما تُكتب القوائم بهذه الطريقة في برامج Prolog. ويعود ذلك إلى أن القوائم تكون أكثر فائدة (ولا تُستخدم عادةً إلا) عندما لا يعرف المبرمج مُسبقًا عدد العناصر التي ستحتويها. فإذا كنا نعلم أن القائمة ستحتوي دائمًا على ثلاثة عناصر، مثل الاسم الأول للشخص ولقبه وجنسيته، فمن الأفضل عمومًا استخدام مُصطلح مُركب compound term بثلاثة مُعاملات three arguments ، مثل

person(john, smith, british).

تُعد القوائم أكثر قيمة عندما لا يُمكن معرفة عدد العناصر المطلوبة مُسبقًا، وقد يختلف ذلك من استخدام لآخر للبرنامج. على سبيل المثال، قد نرغب في تعريف مُسند يقرأ معلومات حول مشتريات مؤسسة في سنة مالية مُحددة، ويكتب قائمة بجميع عناصر أجهزة أو برامج الحاسوب المُشتراة خلال الأشهر من مارس إلى يونيو شاملةً والتي تزيد تكلفتها عن مبلغ مُحدد. في هذه الحالة، لا نرغب بالتأكيد في افتراض أن القائمة ستحتوي دائمًا على خمسة عشر عنصرًا أو أي عدد ثابت آخر من العناصر. قد يكون أي عدد، من الصفر فما فوق.

نحتاج إلى طريقة بديلة لتمثيل قائمة في جملة Prolog، لا تُلزم بعدد عناصرها عند استخدامها. ويتم ذلك باستخدام ترميز "cons" (الذي يُشير إلى مُنشئ القائمة).

في هذا الترميز، تُكتب القائمة بشكل أكثر تعقيدًا من ذي قبل، حيث يربط جزآن معًا برمز الخط العمودي |، (vertical bar) المعروف برمز cons و تُمثل القائمة بالترميز **[elements | list]**.

هذا الرمز | يستخدم لفصل ال head عن ال tail

For example, **[one|[two,three]]** represents **[one,two,three]**.

على سبيل المثال، **[one|[two,three]]** يمثل **[one,two,three]**

يوضح ما يلي بعض الطرق المكافئة لكتابة نفس القائمة المكونة من أربعة عناصر. سننتقل بعد ذلك إلى القوائم ذات الطول المتغير.

```
[alpha,beta,gamma,delta]
[alpha|[beta,gamma,delta]]
[alpha,beta|[gamma,delta]]
[alpha,beta,gamma|[delta]]
[alpha,beta,gamma,delta|[]]
[alpha,beta|[gamma|[delta|[]]]]
```

The cons notation for a list can be used anywhere the 'standard bracketed form' would be valid, e.g.

يمكن استخدام صيغة cons للقائمة في أي مكان يكون فيه "النموذج القياسي بين قوسين" صالحًا، على سبيل المثال

```
?- write([alpha|[beta,gamma,delta]]),nl.
[alpha,beta,gamma,delta]
yes
?- write([alpha,beta,gamma|[delta]]),nl.
[alpha,beta,gamma,delta]
yes
?- write([alpha,beta,gamma,delta|[]]),nl.
[alpha,beta,gamma,delta]
yes
?- write([alpha,beta|[gamma,delta]]),nl.
[alpha,beta,gamma,delta]
yes
?- write([alpha,beta|[gamma|[delta|[]]]]),nl.
[alpha,beta,gamma,delta]
es
```

In the common case where the *elements* part of **[*elements* | *list*]** consists of just one term, the 'cons' notation can be used to construct a list from its head and tail, which are the parts to the left and right of the vertical bar respectively. Thus **[a|[b,c,d]]** denotes the list **[a,b,c,d]**.

في الحالة الشائعة التي يتكون فيها جزء العناصر من [العناصر | القائمة] من مصطلح واحد فقط، يمكن استخدام صيغة "cons" لإنشاء قائمة من رأسها وذيلها، وهما الجزءان على يسار ويمين الشريط الرأسي على التوالي.

As illustrated so far, there would be no benefit gained by using the 'cons' notation rather than the standard bracketed notation. The former is of most value when the ***list*** part is a variable and/or the ***elements*** part contains one or more variables.

كما هو موضح حتى الآن، لن تُجنى أي فائدة من استخدام صيغة "cons" بدلًا من الصيغة القياسية بين قوسين. تُعدّ الصيغة الأولى أكثر فائدة عندما يكون جزء القائمة متغيرًا و/أو يحتوي جزء العناصر على متغير واحد أو أكثر.

For example, if variable **L** is bound to a list, say **[red,blue,green,yellow]**, we can represent a new list with the atom brown inserted before the elements already there by **[brown|L].**

?- L=[red,blue,green,yellow],write([brown|L]),nl.
[brown,red,blue,green,yellow]

If variable *A* is bound to the list **[brown,pink]** and variable *L* is bound to the list **[red,blue,green,yellow]**, the list **[A,A,black|L]** represents **[[brown,pink],[brown,pink],black,red,blue,green,yellow]**.

We are now in a position to write a clause or a query that uses a list b without knowing in advance how many elements it will have. This example shows a new list **L1** created from a list **L** input by the user.

يمكننا الآن كتابة جملة أو استعلام يستخدم list b دون معرفة مسبقة بعدد عناصره. يوضح هذا المثال قائمة جديدة تم إنشاؤها من قائمة L التي أدخلها المستخدم.L1

?- write('Type a list'),read(L),L1=[start|L],write('New list is '),write(L1),nl.

Type a list[[london,paris],[x,y,z],27].

New list is [start,[london,paris],[x,y,z],27]

L = [[london, paris], [x, y, z], 27],

L1 = [start, [london, paris], [x, y, z], 27]

The 'cons' notation for lists is so much more flexible than the standard bracketed notation that some would say that it is the 'correct' notation for lists, and that a list written in the standard bracketed notation, such as **[dog,cat,fish,man]** is just a more human-readable version of **[dog|[cat|[fish|[man|[]]]]]**.

إن صيغة "cons" للقوائم أكثر مرونة بكثير من الصيغة القياسية بين قوسين، لدرجة أن البعض قد يقول إنها الصيغة "الصحيحة" للقوائم، وأن القائمة المكتوبة بالصيغة القياسية بين قوسين، مثل [كلب، قط، سمكة، رجل] هي مجرد نسخة أسهل قراءة من صيغة [كلب|[قط|[سمك|[رجل|[]]]]].

Decomposing a List تحليل القائمة

A common requirement is to perform the same (or a similar) operation on every element of a list. By far the best way of processing a list is to work through its elements one by one from left to right. This can be achieved by breaking the list into its head and tail and processing each separately in a recursive fashion. Paradoxically, breaking a list into its head and tail is often done using the list constructor.

من المتطلبات الشائعة إجراء العملية نفسها (أو عملية مشابهة) على كل عنصر من عناصر القائمة. أفضل طريقة لمعالجة القائمة هي التعامل مع عناصرها واحدًا تلو الآخر من اليسار إلى اليمين. يمكن تحقيق ذلك بتقسيم القائمة إلى رأس وذيل، ومعالجة كلٍّ منهما على حدة بطريقة متكررة. ومن المفارقات أن تقسيم القائمة إلى رأس وذيل غالبًا ما يتم باستخدام مُنشئ القائمة.

The predicate **writeall** defined below writes out the elements of a list, one per line.

The <u>second clause</u> of **writeall** separates a list into its head *A* and tail *L*, writes out *A* and then a newline, then calls itself again recursively. The <u>first clause</u> of **writeall** ensures that evaluation terminates when no further elements of the list remain to be output.

```
/* write out the elements of a list, one per line */
writeall([]).
writeall([A|L]):- write(A),nl,writeall(L).
```

```
?- writeall([alpha,'this is a string',20,[a,b,c]]).
alpha
this is a string
20
[a,b,c]
yes
```

This definition of writeall is typical of many user-defined predicates for list processing. Note that although writeall takes a list as its argument, its definition does not include a statement beginning

هذا التعريف لـ writeall نموذجي للعديد من المسندات المُعرّفة من قِبل المستخدم لمعالجة القوائم. يُرجى ملاحظة أنه على الرغم من أن writeall تأخذ قائمةً كوسيطة لها، إلا أن تعريفها لا يتضمن عبارة تبدأ بـ

```
writeall(L):-
```

Instead, the main part of the definition begins

```
writeall([A|L]):-
```

This makes the definition of the predicate considerably easier.

وهذا يجعل تعريف المسند أسهل إلى حد كبير.

When a goal such as writeall([a,b,c]) is evaluated, it is matched against (unified with) the head of the second clause of writeall. As this is written as writeall([A|L]) the matching process causes A to be bound to atom a and L to be bound to list [b,c]. This makes it easy for the body of the rule to process the head and tail separately.

The recursive call to writeall with the tail of the original list, i.e. L, as its argument is a standard programming technique used in list processing. As is frequently the case, the empty list is treated separately, in this case by the first clause of writeall.

الاستدعاء المتكرر لـ writeall مع ذيل القائمة الأصلية، أي L ، كوسيطة له، هو أسلوب برمجة قياسي يُستخدم في معالجة القوائم. وكما تُعامل القائمة الفارغة بشكل منفصل، وفي هذه الحالة، يتم استخدام الجملة الأولى من writeall

The predicate write_english defined below takes as its argument a list such as

**[[london,england],[paris,france],[berlin,germany],[portsmouth,england],
[bristol,england],[edinburgh,scotland]]**

Each element is a sublist containing the name of a city and the name of the country in which it is located. Calling write_english causes the names of all the cities that are located in England to be output.

كل عنصر عبارة عن قائمة فرعية تحتوي على اسم مدينة واسم الدولة التي تقع فيها. يؤدي استدعاء write_english إلى إخراج أسماء جميع المدن الموجودة في إنجلترا.

يطبع الكود أسماء المدن الإنجليزية من قائمة أزواج المدن والدول، متجاهلاً مدن الدول الأخرى. عند استدعاء الأمر go ، يُفعّل عملية الإخراج، ويعرض فقط المدن التي تُطابق المعايير.

The second clause of **write_english** deals with those sublists that have the atom *england* as their second element. In this case the first element is output, followed by a new line and a recursive call to **write_english**, with the tail of the original list as the argument. Sublists that do not have *england* as their second element are dealt with by the final clause of **write_english**, which does nothing with the sublist but makes a recursive call to **write_english**, with the tail of the original list as the argument.

تتعامل الجملة الثانية من دالة write_english مع القوائم الفرعية التي تحتوي على england كعنصر ثانٍ. في هذه الحالة، يُخرَج العنصر الأول، متبوعًا بسطر جديد واستدعاء متكرر لدالة write_english، مع استخدام ذيل القائمة الأصلية كمعامل. أما القوائم الفرعية التي لا تحتوي على england كعنصر ثانٍ، فتُعالَج بواسطة الجملة الأخيرة من دالة write_english، والتي لا تُجري أي شيء مع القائمة الفرعية، بل تُجري استدعاءً متكررًا لدالة write_english، مع استخدام ذيل القائمة الأصلية كمعامل.

```
write_english([]).
write_english([[City,england]|L]):-
   write(City),nl,
   write_english(L).
write_english([A|L]):-write_english(L).
go:- write_english([[london,england],[paris,france],
    [berlin,germany],[portsmouth,england],
    [bristol,england],
    [edinburgh,scotland]]).
```

Base Case:
write_english([]).

This predicate defines the base case for the recursion. It states that if the list is empty ([]), there’s nothing to write, and the predicate succeeds.

Recursive Rule for English Cities:

```
write_english([[City, england] | L]) :-
    write(City), nl,
    write_english(L).
```

This rule matches a non-empty list where the head is a list containing a city and the term "england".

City is extracted from this head.

write(City) prints the name of the city.

nl moves the output to the next line.

The predicate then calls itself recursively with the tail L, allowing it to handle remaining elements.

Recursive Rule for Other Elements:

```
write_english([A | L]) :- write_english(L).
```

This rule handles any other elements that do not match the specific pattern of a city in England.

It simply discards the head (A) and continues checking the tail (L).

This allows the program to ignore cities that don’t fit the [City, england] format.

Goal Predicate:

The go predicate serves as the entry point to execute the program. When go is called, it invokes write_english with a predefined list of cities. The list contains pairs of cities and their corresponding countries. Only cities in England will be printed due to the structure of the recursive rules.

```
?- go.
london
portsmouth
Bristol
Yes
```

The predicate **replace** defined below takes as its first argument a list of at least one element. If the second argument is an unbound variable, it is bound to the same list with the first element replaced by the atom **first**. Using the 'cons' notation, the definition takes only one clause.

```
replace([A|L],[first|L]).
```

Predicate Definition

Purpose: This predicate serves to replace the first element of a given list with the atom first.

Structure:

Input: The first argument is a list of at least one element denoted as [A | L], where A is the first element and L is the rest of the list.

Output: The second argument is a new list where the first element (A) is replaced by first, while the rest of the list (L) remains unchanged.

How It Works

Pattern Matching: When the replace predicate is called, it uses pattern matching to decompose the first list into its head (A) and tail (L).

Replacement: It constructs a new list with first as the head and the unchanged tail (L).

Binding: If the second argument (L in the example) is an unbound variable (i.e., it hasn't been assigned a value yet), Prolog binds it to the newly constructed list.

First Example:

?- replace([1,2,3,4,5], L).

Input: A list [1, 2, 3, 4, 5].

Output: L = [first, 2, 3, 4, 5].

Explanation: The first element 1 is replaced with first, and the rest of the list ([2, 3, 4, 5]) remains the same.

Second Example:

?- replace([[a,b,c],[d,e,f],[g,h,i]], L).

Input: A list of lists [[a,b,c],[d,e,f],[g,h,i]].

Output: L = [first, [d, e, f], [g, h, i]].

Explanation: Similar to the first example, the first element [a,b,c] is replaced by first, while the rest of the list ([[d,e,f],[g,h,i]]) remains unchanged.

Built-in Predicate: member

The ability to represent data in the form of lists is such a valuable feature of Prolog that several built-in predicates have been provided for it. The most commonly used of these are described in this and the following sections.
The **member** built-in predicate takes two arguments. If the first argument is any term except a variable and the second argument is a list, **member** succeeds if the first argument is a member of the list denoted by the second argument (i.e. one of its list elements).
تُعدّ القدرة على تمثيل البيانات على شكل قوائم ميزة قيّمة في لغة برولوغ، لدرجة أنها زوّدت بالعديد من المسندات المُدمجة. يُوضّح هذا القسم والأقسام التالية أكثر هذه المسندات شيوعًا.

يأخذ المسند المُدمج العضو وسيطتين. إذا كان الوسيط الأول أي مصطلح باستثناء متغير، وكان الوسيط الثاني قائمة، ينجح العضو إذا كان الوسيط الأول عضوًا في القائمة التي يُشير إليها الوسيط الثاني (أي أحد عناصر قائمتها).

?- member(a,[a,b,c]).
yes
?- member(mypred(a,b,c),[q,r,s,mypred(a,b,c),w]).
yes
?- member(x,[]).
no
?- member([1,2,3],[a,b,[1,2,3],c]).

yes

If the first argument is an unbound variable, it is bound to an element of the list working from left to right (thus if it is called only once it will be bound to the first element). This can be used in conjunction with backtracking to find all the elements of a list in turn from left to right, as follows.

إذا كانت الوسيطة الأولى متغيرًا غير مرتبط، فسيتم ربطه بعنصر من القائمة يعمل من اليسار إلى اليمين (وبالتالي، إذا تم استدعاؤه مرة واحدة فقط، فسيتم ربطه بالعنصر الأول). يمكن استخدام هذا بالتزامن مع التتبع العكسي للعثور على جميع عناصر القائمة بالترتيب من اليسار إلى اليمين، كما يلي.

?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no

Predicate get_answer2 defined below reads a term entered by the user. It loops using repeat, until one of a list of permitted answers (yes, no or maybe) is entered and the member goal is satisfied.

```
get_answer2(Ans):-repeat,
    write('answer yes, no or maybe'),read(Ans),
    member(Ans,[yes,no,maybe]),
    write('Answer is '),write(Ans),nl,!.
```

?- get_answer2(X).
answer yes, no or maybe: possibly.
answer yes, no or maybe: unsure.
answer yes, no or maybe: maybe.
answer is maybe
X = maybe

Built-in Predicate: length

The length built-in predicate takes two arguments. The first is a list. If the second is an unbound variable it is bound to the length of the list, i.e. the number of elements it contains.

يأخذ المسند المضمن "الطول" وسيطتين. الأولى قائمة. إذا كانت الثانية متغيرًا غير مرتبط، فهي مرتبطة بطول القائمة، أي عدد عناصرها

```
?- length([a,b,c,d],X).
X = 4
?- length([[a,b,c],[d,e,f],[g,h,i]],L).
L = 3
?- length([],L).
L = 0
```

If the second argument is a number, or a variable bound to a number, its value is compared with the length of the list.

إذا كان الجزء الثاني من القائمة عبارة عن رقم، أو متغير مرتبط برقم، تتم مقارنة قيمته بطول القائمة.

```
?- length([a,b,c],3).
yes
?- length([a,b,c],4).
no
?- N is 3,length([a,b,c],N).
N = 3
```

Built-in Predicate: reverse

The **reverse** built-in predicate takes two arguments. If the first is a list and the second is an unbound variable (or vice versa), the variable will be bound to the value of the list with the elements written in reverse order, e.g.

يأخذ المسند العكسي المدمج وسيطتين. إذا كان الأول قائمة والثاني متغيرًا غير مرتبط (أو العكس)، فسيتم ربط المتغير بقيمة القائمة مع كتابة عناصرها بترتيب عكسي، على سبيل المثال:

```
?- reverse([1,2,3,4],L).
```

L = [4,3,2,1]
?- reverse(L,[1,2,3,4]).
L = [4,3,2,1]
?- reverse([[dog,cat],[1,2],[bird,mouse],[3,4,5,6]],L).
L = [[3,4,5,6],[bird,mouse],[1,2],[dog,cat]]

Note that the order of the elements of the sublists [dog,cat] etc. is not reversed.
If both arguments are lists, reverse succeeds if one is the reverse of the other.

لاحظ أن ترتيب عناصر القوائم الفرعية [كلب، قط] وما إلى ذلك ليس معكوسًا.
ينجح العكس إذا كانت إحداهما معكوسة للأخرى.

?- reverse([1,2,3,4],[4,3,2,1]).
yes

?- reverse([1,2,3,4],[3,2,1]).
no

The predicate **front/2** defined below takes a list as its first argument. If the second argument is an unbound variable it is bound to a list which is the same as the first list with the last element removed. For example if the first list is [a,b,c], the second will be [a,b]. In the body of the rule the first list L1 is reversed to give L3. Its head is then removed to give L4 and L4 is then reversed back again to give L2.

```
front(L1,L2):-
   reverse(L1,L3),remove_head(L3,L4),reverse(L4,L2).
remove_head([A|L],L).
```

?- front([a,b,c],L).
L = [a,b]
?- front([[a,b,c],[d,e,f],[g,h,i]],L).
L = [[a,b,c],[d,e,f]]
The front predicate can also be used with two lists as arguments. In this case it tests whether the second list is the same as the first list with the last element removed.

```
?- front([a,b,c],[a,b]).
yes
?- front([[a,b,c],[d,e,f],[g,h,i]],[[a,b,c],[d,e,f]]).
yes
?- front([a,b,c,d],[a,b,d]).
No
```

Built-in Predicate: append

The term *concatenating* two lists means creating a new list, the elements of which are those of the first list followed by those of the second list. Thus concatenating **[a,b,c]** with **[p,q,r,s]** gives the list **[a,b,c,p,q,r,s]**. Concatenating **[]** with **[x,y]** gives **[x,y]**.

The **append** built-in predicate takes three arguments. If the first two arguments are lists and the third argument is an unbound variable, the third argument is bound to a list comprising the first two lists concatenated, e.g.

```
?- append([1,2,3,4],[5,6,7,8,9],L).
L = [1,2,3,4,5,6,7,8,9]
?- append([],[1,2,3],L).
L = [1,2,3]
?- append([[a,b,c],d,e,f],[g,h,[i,j,k]],L).
L = [[a,b,c],d,e,f,g,h,[i,j,k]]
```

The append predicate can also be used in other ways. When the first two arguments are variables and the third is a list it can be used with backtracking to find all possible pairs of lists which when concatenated give the third argument, as follows.

```
?- append(L1,L2,[1,2,3,4,5]).
L1 = [] ,
L2 = [1,2,3,4,5] ;
L1 = [1] ,
L2 = [2,3,4,5] ;
L1 = [1,2] ,
L2 = [3,4,5] ;
L1 = [1,2,3] ,
L2 = [4,5] ;
L1 = [1,2,3,4] ,
L2 = [5] ;
L1 = [1,2,3,4,5] ,
L2 = [] ;
```

```
no
```

This example shows a list broken up in a more complex way.

```
?- append(X,[Y|Z],[1,2,3,4,5,6]).
X = [] ,
Y = 1 ,
Z = [2,3,4,5,6] ;
X = [1] ,
Y = 2 ,
Z = [3,4,5,6] ;

X = [1,2] ,
Y = 3 ,
Z = [4,5,6] ;
X = [1,2,3] ,
Y = 4 ,
Z = [5,6] ;
X = [1,2,3,4] ,
Y = 5 ,
Z = [6] ;
X = [1,2,3,4,5] ,
Y = 6 ,
Z = [] ;
No
```

List Processing: Examples

This section shows some examples of list processing, all of which illustrate the use of recursion.

Example 1

The predicate find_largest/2 takes a list of numbers as its first argument and assigns the value of the largest element to its second argument (assumed to be an unbound variable). It is assumed that the list contains at least one number.

```
find_largest([X|List],Maxval):-
find_biggest(List,Maxval,X).
find_biggest([],Currentlargest,Currentlargest).
find_biggest([A|L],Maxval,Currentlargest):-
   A>Currentlargest,
   find_biggest(L,Maxval,A).
find_biggest([A|L],Maxval,Currentlargest):-
   A=<Currentlargest,
   find_biggest(L,Maxval,Currentlargest).
```

Calling the find_largest goal with the list of numbers as its first argument causes the first element of the list to be removed and passed to find_biggest as its third argument (the largest number found so far). The remainder of the list is passed to find_biggest as its first argument (a list of the numbers not yet examined). The second argument of find_largest (Maxval) represents the overall largest number and is unbound because the value is not yet known. It is passed to find_biggest as its second argument.
The three arguments of find-biggest/3 represent in order:
• the list of numbers not so far examined
• the overall largest number (to be passed back to find_largest)
• the largest number found so far.
So the first clause of find_biggest can be read as: 'if there are no more numbers remaining unexamined, return the largest number so far (third argument) as the value of the overall largest number (second argument)'. The second argument becomes bound.
The second clause of find_biggest can be read as: 'if the list of numbers so far not examined begins with value A and A is larger than the largest so far found, call find_biggest (recursively) with L, the tail of the (first argument) list, as the new first argument (list of unexamined numbers) and with A as the third argument (largest number so far)'. The second argument (Maxval) is unbound.
The final clause of find_biggest can be read as: 'if the list of numbers so far not examined begins with value A and A is not larger than the largest so far found, call find_biggest (recursively) with L, the tail of the (first argument) list, as the new first argument (list of unexamined numbers) and with the third argument (largest number so far) unchanged'. The second argument (Maxval) is unbound.

?- find_largest([10,20,678,-4,-12,102,-5],M).
M = 678
?- find_largest([30,10],M).
M = 30
?- find_largest([234],M).
M = 234

Example 2
The front/2 predicate was defined in Section 9.6 as an example of the use of the reverse built-in predicate. It takes a list as its first argument. If the second argument is an unbound variable it is bound to a list which is the same as the first list with the last element removed. For example if the first list is [a,b,c], the second will be [a,b].

The predicate can be defined more efficiently using recursion as follows.

```
front([X],[]).
front([X|Y],[X|Z]):-front(Y,Z).
```

The two clauses can be read as 'the front of a list with just one element is the empty list' and 'the front of a list with head X and tail Y is the list with head X and tail Z where Z is the front of Y', respectively.

?- front([alpha],L).
L = []
?- front([alpha,beta,gamma],LL).
LL = [alpha,beta]
?- front([[a,b],[c,d,e],[f,g,h]],L1).
L1 = [[a,b],[c,d,e]]

Example 3
One area in which different Prolog implementations can vary considerably is the provision of built-in predicates for list processing. If your implementation does not have **member/2**, **reverse/2** or **append/3** (described in Sections 9.4, 9.6 and 9.7 respectively) you can define your own with just a few clauses as shown below.

```
member(X, [X|L]).
member(X, [_|L]):-member(X,L).
reverse(L1,L2):-rev(L1,[],L2).
rev([],L,L).
rev([A|L],L1,L2):-rev(L, [A|L1],L2).
append([],L,L).
append([A|L1],L2,[A|L3]):-append(L1,L2,L3).
```

The two clauses defining **member/2** just state that *X* is a member of any list with head *X* (i.e. that begins with *X*) and that *X* is a member of any list for which it is not the head if it is a member of the tail.
The definitions of the other two predicates are slightly more complex and are left without explanation.
If your implementation of Prolog has any or all these predicates built-in, attempting to load the above program will cause an error, as it is not permitted to attempt to redefine a built-in predicate.

However the definitions can be tested by renaming the predicates systematically as mymember, myreverse and myappend, say, giving the following program.

```
mymember(X, [X|L]).
mymember(X, [_|L]):-mymember(X,L).
myreverse(L1,L2):-rev(L1,[],L2).
rev([],L,L).
rev([A|L],L1,L2):-rev(L, [A|L1],L2).
myappend([],L,L).
myappend([A|L1],L2,[A|L3]):-myappend(L1,L2,L3).
```

This can then be tested using some of the examples in Sections 9.4, 9.6 and 9.7, for which it gives the same results in each case.

```
?- mymember(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no
?- mymember(mypred(a,b,c),[q,r,s,mypred(a,b,c),w]).
yes
?- mymember(x,[]).
no
?- myreverse([1,2,3,4],L).
L = [4,3,2,1]
?- myreverse([[dog,cat],[1,2],[bird,mouse],[3,4,5,6]],L).
L = [[3,4,5,6],[bird,mouse],[1,2],[dog,cat]]
?- myappend([1,2,3,4],[5,6,7,8,9],L).
L = [1,2,3,4,5,6,7,8,9]
?- myappend([],[1,2,3],L).
L = [1,2,3]
```

Using findall/3 to Create a List

It would often be desirable to find all the values that would satisfy a goal, not just one of them. The **findall/3** predicate provides a powerful facility for creating lists of all such values. It is particularly useful when used in conjunction with the Prolog database.

findall(Template, Goal, List).
Template – what you want to collect

Goal – the condition or pattern to satisfy

List – where the results go

If the database contains the five clauses

```
person(john,smith,45,london).
person(mary,jones,28,edinburgh).
person(michael,wilson,62,bristol).
person(mark,smith,37,cardiff).
person(henry,roberts,23,london).
```

a list of all the surnames (the second argument of **person**) can be obtained using **findall** by entering the goal:
?- findall(S,person(_,S,_,_),L).
This returns
L = [smith,jones,wilson,smith,roberts]

L is a list of all the values of variable *S* which satisfy the goal **person(_,S,_,_)**.
The predicate **findall/3** has three arguments. The first is generally an unbound variable, but can be any term with at least one unbound variable as an argument (or equivalently any list with at least one unbound variable as a list element).
The second argument must be a goal, i.e. must be in a form that could appear on the right-hand side of a rule or be entered at the system prompt.
The third argument should be an unbound variable. Evaluating **findall** will cause this to be bound to a list of all the possible values of the term (first argument) that satisfy the goal (second argument).
More complex lists can be constructed by making the first argument a term involving several variables, rather than using a single variable. For example

?- findall([Forename,Surname],person(Forename,Surname,_,_),L).
returns the list
L = [[john,smith],[mary,jones],[michael,wilson],[mark,smith],[henry,roberts]]

The term (first argument) can be embellished further, e.g.
?- findall([londoner,A,B],person(A,B,_,london),L).

returns the list
L = [[londoner,john,smith],[londoner,henry,roberts]]

Given a database containing clauses such as

```
age(john,45).
age(mary,28).
age(michael,62).
age(henry,23).
age(george,62).
age(bill,17).
age(martin,62).
```

the predicate **oldest_list/1** defined below can be used to create a list of the names of the oldest people in the database (in this case michael, george and martin, who are all 62).
It begins by calling **findall** to find the ages of all the people in the database and put them in a list *Agelist*.
It then uses the predicate **find_largest** (defined previously) to find the largest of these values and bind variable *Oldest* to that value. Finally, it uses **findall** again to create a list of the names of all the people of that age.

```
oldest_list(L):-
   findall(A,age(_,A),Agelist),
   find_largest(Agelist,Oldest),
   findall(Name,age(Name,Oldest),L).
```

?- oldest_list(L).
L = [michael,george,martin]
The final example in this section shows a predicate find_under_30s used in conjunction with the age predicate from the previous example to create a list of the names of all those people under 30. This requires only one new clause.

```
find_under_30s(L):-findall(Name,(age(Name,A),A<30),L).
```

?- find_under_30s(L).
L = [mary,henry,bill]

The second argument of **findall** is the goal **(age(Name,A),A<30)**. It is important to place parentheses around it so that it is treated as a single (compound) goal with two component subgoals, not as two goals. Omitting the parentheses would give a predicate named **findall** with four arguments. This would be an entirely different predicate from **findall/3** and one unknown to the Prolog system.

Practical Exercise

(1) Define and test a predicate **pred1** that takes a list as its first argument and returns the tail of the list as its second argument, e.g.

```
?- pred1([a,b,c],L).
L = [b,c]
```

(2) Define and test a predicate **inc** that takes a list of numbers as its first argument and returns a list of the same numbers all increased by one as its second argument, e.g.

```
?- inc([10,20,-7,0],L).
L = [11,21,-6,1]
```

(3) Define and test a predicate **palindrome** that checks whether a list reads the same way forwards and backwards, e.g.

```
?- palindrome([a,b,c,b,a]).
yes
?- palindrome([a,b,c,d,e]).
No
```

(4) Define and test a predicate **putfirst** that adds a specified term to the beginning of a list, e.g.

```
?- putfirst(a,[b,c,d,e],L).
L = [a,b,c,d,e]
```

(5) Define and test a predicate **putlast** that adds a specified term to the end of a list,
e.g.

```
?- putlast(e,[a,b,c,d],L).
L = [a,b,c,d,e]
```

(6) Using **findall** define and test predicates **pred1/2**, **pred2/2** and **pred3/2** that modify a list, as shown in the following examples:

?- pred1([a,b,c,d,e],L).
L = [[a],[b],[c],[d],[e]]

?- pred2([a,b,c,d,e],L).
L = [pred(a,a),pred(b,b),pred(c,c),pred(d,d),pred(e,e)]

?- pred3([a,b,c,d,e],L).
L = [[element,a],[element,b],[element,c],[element,d],[element,e]]

Explanation of the example in the list

```
member(X,[X|L]).
member(X,[_|L]):-member(X,L).
reverse(L1,L2):-rev(L1,[],L2).
rev([],L,L).
rev([A|L],L1,L2):-rev(L,[A|L1],L2).
append([],L,L).
append([A|L1],L2,[A|L3]):-append(L1,L2,L3).
```

☑ 1. member(X, List)

member(X, [X|L]).

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

□ The first clause: X is the **head** of the list → success.

□ The second clause: if **X ≠ head**, skip the head, and check the **tail** recursively.

🔍 Example:

?- member(3, [1, 2, 3, 4]).

Step-by-step:

1. List is [1,2,3,4], head is 1 ≠ 3 → try member(3, [2,3,4])
2. Head is 2 ≠ 3 → try member(3, [3,4])
3. Head is 3 == 3 → then success ☑

☑ **2. reverse(L1, L2)**

Reverses L1 into L2 using an accumulator (the second argument in rev).

reverse(L1, L2) :- rev(L1, [], L2).

rev([], L, L).

rev([A|L], L1, L2) :- rev(L, [A|L1], L2).

صُمم مسند العكس (reverse) لأخذ قائمة (L1) وإنتاج نسخة معكوسة منها (L2) ولتحقيق ذلك، يستخدم مسندًا مساعدًا rev ، الذي يُجري عملية الانعكاس الفعلية باستخدام مُراكم (accumulator).

reverse(L1, L2) :- rev(L1, [], L2).

- L1: The input list that we want to reverse.
- L2: The output list that will hold the reversed version of L1.

- **Functionality**: It calls the helper predicate rev, passing:
  - L1 (the list to reverse),
  - An empty list [] as an initial value for the accumulator (this will build the reversed list),
  - L2, which will store the final reversed list once the recursion completes.

rev([], L, L).

- When rev is called with an **empty list []**, it means that all elements have been processed.
- Here, L is the accumulated list that has been built up during the recursion. The predicate succeeds and returns L as the final output.
- The important thing is that when the input list is empty, the second and third arguments (L and L) are the same, meaning that whatever is in L gets returned as the result.

عند استدعاء rev بقائمة فارغة []، فهذا يعني أن جميع العناصر قد تمت معالجتها.

في هذه الحالة، L هي القائمة المتراكمة التي تم إنشاؤها أثناء التكرار. ينجح المسند ويُرجع L كمخرج نهائي.

المهم هو أنه عندما تكون قائمة الإدخال فارغة، يكون الوسيطان الثاني والثالث L وLمتماثلين، مما يعني أن كل ما هو موجود في L يُرجع كنتيجة.

rev([A | L], L1, L2) :- rev(L, [A | L1], L2).

- **Parameters**:
  - [A | L]: This pattern matches a non-empty list, where A is the head (the first element) and L is the tail (the rest of the list).
  - L1: The current state of the accumulator that holds the reversed elements.
  - L2: The target variable where the reversed list will be returned.

[A | L] يتطابق هذا النمط مع قائمة غير فارغة، حيث A هو الرأس (العنصر الأول) و Lهو الذيل (بقية القائمة).

L1 الحالة الحالية للمجمع (accumulator) الذي يحمل العناصر المعكوسة.

L2 المتغير المستهدف الذي سيتم إرجاع القائمة المعكوسة إليه.

- **Functionality**:
  - The head element A is added to the front of the accumulator: [A | L1].
  - The recursion proceeds with the tail L (the rest of the original list). The updated state of the accumulator is now [A | L1] for the next recursive call.
  - This pattern continues until the entire original list has been processed. Each call pushes the current head onto the accumulator, effectively reversing the order.

يُضاف عنصر الرأس A إلى مقدمة المُراكم [A | L1] :

يستمر التكرار مع الذيل L (باقي القائمة الأصلية). تصبح الحالة المُحدّثة للمُراكم الآن [A | L1] للاستدعاء التكراري التالي.

يستمر هذا النمط حتى تتم معالجة القائمة الأصلية بالكامل. كل استدعاء يدفع الرأس الحالي إلى المُراكم، مما يعكس الترتيب فعليًا.

تتبع الكود يكون بالشكل التالي:

1. **Initial Call:**
   - reverse([1, 2, 3], L2): Calls rev([1, 2, 3], [], L2)
2. **First Recursive Call:**
   - rev([1, 2, 3], [], L2):
     - Matches A = 1 and L = [2, 3]
     - Makes the call rev([2, 3], [1], L2)
3. **Second Recursive Call:**
   - rev([2, 3], [1], L2):
     - Matches A = 2 and L = [3]
     - Makes the call rev([3], [2, 1], L2)
4. **Third Recursive Call:**
   - rev([3], [2, 1], L2):
     - Matches A = 3 and L = []
     - Makes the call rev([], [3, 2, 1], L2)
5. **Base Case:**
   - rev([], [3, 2, 1], L2):
     - Base case is reached, so it returns [3, 2, 1] as L2.

Notes

- The reverse/2 predicate is an efficient way to reverse a list using tail recursion.
- An **accumulator** (L1) is used to collect the reversed elements during the recursive calls.
- The process continues to build up the list from the last element to the first until the original list is fully traversed.
- When the base case is reached, the accumulated list is returned as the reverse of the original list.

🔍 Example:

?- reverse([a,b,c], R).

Step-by-step:

1. reverse([a,b,c], R) calls rev([a,b,c], [], R)
2. rev([a,b,c], [], R) → calls rev([b,c], [a], R)
3. rev([b,c], [a], R) → calls rev([c], [b,a], R)
4. rev([c], [b,a], R) → calls rev([], [c,b,a], R)
5. Base case: rev([], [c,b,a], R) → R = [c,b,a]

?- reverse([1,2,3], L2).

Step 1:

reverse([1,2,3], L2)
→ calls rev([1,2,3], [], L2)

Step 2:

rev([1,2,3], [], L2)
→ head = 1, tail = [2,3]
→ calls rev([2,3], [1], L2)

Step 3:

rev([2,3], [1], L2)
→ head = 2, tail = [3]
→ calls rev([3], [2,1], L2)

Step 4:

rev([3], [2,1], L2)
→ head = 3, tail = []
→ calls rev([], [3,2,1], L2)

Step 5:

rev([], [3,2,1], L2)

→ base case → L2 = [3,2,1]

🎉 Final Result:

?- reverse([1,2,3], L2).
L2 = [3,2,1].

☑ 3. append(L1, L2, Result)

append([], L, L).

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

Appends two lists: L1 and L2 into Result.

- **Base case:**

append([], L, L).

- If the first list is empty ([]), then appending it to L just results in L.

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

- *If the first list is* [A|L1] *(<u>head</u>* <u>A</u> *<u>and tail</u>* <u>L1</u>*), then the result starts with* A*, followed by the appending of the <u>tail</u>* <u>L1</u> *and <u>list</u>* <u>L2.</u>

  The predicate takes two lists and produces their concatenation.

  It works by removing the head of the first list and recursively appending the remaining tail to the second list, rebuilding the result step-by-step.

☑ Example: append([1,2], [3,4], R).

It matches the recursive rule:

A=1, L1=[2], L2=[3,4], Result=[1|L3]

then recurse with:

append([2], [3,4], L3)

Again, recursive step:

A=2, L1=[], L2=[3,4], L3=[2|L4]

recurse with:

append([], [3,4], L4)

Base case:

L4=[3,4] (since when first list is empty, result is the second list).

So

L4=[3,4]

L3=[2|L4]=[2,3,4]

Result=[1|L3]=[1,2,3,4]

?- reverse([1,2,3], Rev), append(Rev, [4,5], Result), member(2, Result).

1. Reverse [1,2,3] → Rev = [3,2,1]
2. Append [3,2,1] + [4,5] → Result = [3,2,1,4,5]
3. Check if 2 is a member → yes ☑