Lecture 7

Prolog

Now we know how to reason with logic. But when it comes to coding conventional languages are too tedious.

We will see a popular alternative: Prolog.

Reference:

- Textbook: Section 9.4, pp 289–295.
- GNU Prolog documentation: within our department server (e.g., virtue), `/usr/local/GNU/gprolog-1.2.1/gprolog-1.2.1/doc/manual.ps`

Why C++ is not good enough for our task

Can we write a **chaining system in C++**? We have two choices.

1. We can **represent clauses as objects** in our program. We can have a knowledge base which is a database of these clauses, etc. Our program would read these clauses from a file and reason with them.

   But… it is slow. Everytime we need to deduce a fact, we have to look through the database to find the relevant clauses.
   The implication form clauses are fixed… can we do better?

2. We can **represent clauses as functions** in our program. Each function either recursively call other functions or directly return some object which satisfy the clause.

   But… it is tedious. We have to deal with all procedural aspects like back-tracking in order to write a function.
   The next page shows why we don't like this approach…

Let's see an example

Let's look at the second approach and see why we say it's tedious.

- Suppose we want to represent the following clauses in a function:
  $Mother(x, y) \Rightarrow Parent(x, y)$
  $Father(x, y) \Rightarrow Parent(x, y)$
- The function Parent(x,y) does the following:
  1. See if any $Parent(x, y)$ pair is stored in the KB directly.
  2. Call $Mother(x, y)$. If it output anything, report.
  3. Call $Father(x, y)$. If it output anything, report.
  4. We want to report **one result at a time**, so that it will not waste a lot of time if we just need one solution.
- This is tedius since every function we must do the same things.

Prolog: overview

- Prolog: **generate functions automatically**. Our program contains only **clauses** like `parent(X) :- father(X); mother(X).`.
- A "procedure" like `parent` is defined by clauses. Prolog uses **depth first**, **backward chaining** to work on them.
  But programmers can affect the chaining slightly.
- The procedure does **unification** automatically, although the **occurrence check** is left out. So sometimes unification causes Prolog to "hang".
  But it is rare to have such problem.
- When there are more than one solution, the **first** answer is returned, but the caller has the option to "reject" the answer and ask the procedure to **continue** finding other solutions.
  So an procedure invocations may give you a "continuation".

Running a Prolog interpretor

In this course we will write Prolog programs and use a **Prolog interpretor** called GNU Prolog ("gprolog") to read it.

Its RPM (for Linux) and setup EXE (for Windows) can be found in the course web page which you can download and install to your computer.

Or in `virtue`, add `/usr/local/GNU/gprolog-1.2.1/bin` to your search path (usually specified in the text file `.profile` or `.cshrc` in your server home directory). (You also need to set path in Windows.)

After setup, you can run it creating a terminal (DOS prompt in Windows) and type `gprolog`:

```
>gprolog
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2003 Daniel Diaz
| ?-
```


Atomic terms

Prolog has two types of data, or **terms**. The basic terms are called **atomic** terms: **atoms**, **integers**, and **real numbers**.

- An **atom** is what other language would call a string. If it starts with a lower case letter, and contains only only digits, letters and underscores, then it can be written directly. Otherwise, use **single quotes**. E.g.:

  `june`, `year_2001`, `'MyCar'`, `'2001/9/11'`, etc.
  As in C, it may contain escape sequences, like `'Hello, World\n'`.
- An **integer** and a **real number** is exactly the same thing that you learn in any other language, e.g.,

  Integers: `1`, `2`, `-5`
  Real numbers: `3.14`, `-2.5`, `6.02e23`

  Remember that we are in the context of First-Order logic reasoning, so "objects" has the meaning of first-order logic objects.

Structures

The other type of data is called **structures**, and is in the form

```
functor(arg1, arg2, arg3, …)
```

The `functor` must be an atom; and each `argN` can be any term: a structure or an atom. Each structure has **one or more** arguments.

Never none.

So you can have `date(2004, 3, 25)`, `time(4, 30, pm)`, or `date(2004, 3, 25, time(4, 30, pm))`, or even `S(S(S(0)))`.

Number of arguments of the structure is called its **arity**. In general, structures with the same name but different arity can be considered unrelated.

Predicates

- A **predicate** is defined by a term, identified by the functor and arity. A predicate with no argument is written like an atom.
- There are some **built-in** predicate. E.g.,

```
atom(isaac)      %% 'isaac' is an atom (yes)
float(3.14159)   %% 3.14159 is floating point (yes)
integer(123)     %% 123 is an integer (yes)
true             %% always true
```

  `%` is the comment character in Prolog, although you can also use `/*` and `*/`.

- A predicate forms a simple **clause** by following it by a period (.). Then it can be directly typed into the Prolog interpretor as a **query**:

```
| ?- atom(isaac).
yes
```

- Each predicate has its **procedure** defining it. We refer to a procedure by a **procedure indicator**, like `atom/1` or `true/0`.

Knowledge base

Prolog keeps a KB containing a **list of clauses** for each procedure.

We can change the KB dynamically with **asserta** and **assertz** (adding the new clause at beginning and end of the list respectively).

```
| ?- assertz(parent(fred,cindy)).     %% Add as last clause of parent/2
yes
| ?- assertz(parent(mary,cindy)).
yes
| ?- parent(fred,cindy).              %% Query
yes                                   %% Means entailed by KB
| ?- parent(fred, mary).
no                                    %% Means not entailed by KB
```

The output is simplified a bit to save space.

The `assertz` line looks like a query as well… In fact, it **is** a query (and `assertz` is a built-in predicate)! So you can see the "yes" answer.

So a structure can be the argument of a predicate.

The query has the **side effect**, changing the KB upon execution.

Some other predicates with side effects

- `retract/1`: remove a clause from KB. E.g., `retract(parent(fred, cindy)).` removes the first such clause of `parent/2`.
- `abolish/1`: throw away the procedure. E.g., `abolish(parent/2).` removes the `parent/2` procedure completely.
- `consult/1`: load and "byte-compile" a file to KB: `consult(test)` looks for the file `test.pl`, which should contain clauses terminated by periods. They replace the current procedure definitions . One may also use `[test].`
  We usually think of such a file as a "program". Byte-compile makes it faster. There is a command (`gplc`) to "compile" it, making it even faster.
- `write/1`: print its argument on the screen: `write(hello).` prints the word "hello" (without anything extra).
- `system/1`: run an external program: `system(ls).` shows the files in the current directory on the screen by executing `ls`.

Dynamic clauses

If you put

```
parent(fred, cindy).
```

into a file and load it using `consult`, and try to use `assertz` to modify `happy` afterwards, you get this:

```
| ?- assertz(parent(fred, cindy)).
uncaught exception: error(permission_error(modify,\
static_procedure,happy/1),assertz/1)
```

Compiled procedures cannot be modified using `assertz` and `retract`. If you need to do them, you must keep them un-compiled with `dynamic`:

```
:- dynamic(parent/2).
parent(fred, cindy).
```

In this way, you can avoid having to enter clauses every time you run `gprolog`, but still allow the clauses be changed on run-time.

Price: it is not as fast as compiled code.

Horn clauses

Horn clauses is a head followed by a conjunction of simple clauses:

$$P(A) \land Q(A) \land R(A) \Rightarrow S(A)$$

But either side can be empty.

In Prolog, procedure can contain Horn clauses is written like this:

```
s(a):- p(a), q(a), r(a).
```

Seems like new syntax, but actually it is also a "structure"! More about it later…

In the above sentence, **comma means "and"**. We also have "**or**", which is denoted by a **semicolon**:

```
s(a):- p(a), q(a); r(a).
```

Comma has higher precedence than semicolon, so the above means "(p(a) and q(a)) or r(a)". This can be overridden by parentheses.

A simple example

If our program contains

```
a(1).
a(2).
a(3):-a(1); a(2).
a(4):-a(1).
a(4):-a(2).
a(5):-a(3), a(4).
```

`a(0)` gives "no", `a(1)` and `a(2)` gives "yes"…

`a(3)` and `a(4)` gives "yes" twice, and `a(5)` gives "yes" 4 times!
We will see why we want multiple answers once we know about Prolog variables.

Each time Prolog shows this:

```
true ?
```

Typing `;` gives one more solution, `a` gives all solution, and Enter stops.

Execution model

Prolog **process a query** by using it as the **current goal** to be proved:

1. If the current goal is a **OR** clause: for each component from left to right, recursively prove the component as a **sub-goal**. The answers of each sub-goal is also become an answer to the current goal.
2. For **AND** clauses: recursively for the left-most component, recursively find ways that the component can be proved. For each of the ways, check the next component similarly. Thus each answer of the AND clause is a way to prove all its component. These answers are returned one at a time.
3. If it is a **predicate**, find clauses in KB with a head that matches the query. They are tried one by one to find all answers (just like they are "or"-ed together). If it is a simple clause, conclude it is proved; if it is an implication, recursively find each way that the body can be proved.

Example

Suppose the KB contains this…

```
saturday(today).
sunday(tomorrow).
holiday(today):- write('Check Sunday\n'), sunday(today).
holiday(today):- saturday(today), write('With reservation\n').
happy(fiona):- holiday(today), write('not bad\n'), sunday(today).
happy(isaac):- happy(fiona); holiday(today), write('holiday!').
```

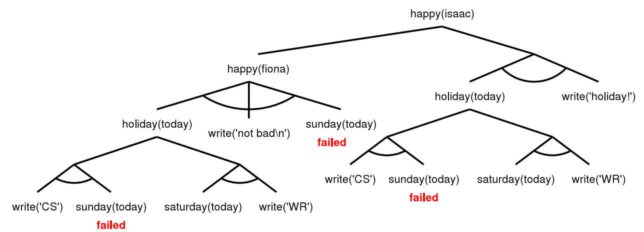
What will happen if we query `happy(isaac)`?

```
| ?- happy(isaac).
Check Sunday
With reservation
not bad
Check Sunday
With reservation
holiday!

yes
```


And-or graph

The computation is sometimes drawn as an AND-OR graph by using branches for OR, and branches with arcs for AND. E.g.,:



But it grows too large pretty quickly, so it is not really that useful.

My advice: use it only for understanding execution model, and use basic programming techniques like modularity to keep things organized.

Prolog variables

- A variable looks similar to **atoms without quotes**, except that it begins with a **capital letter**. E.g., `X`, `B2`, `First_Queen`, etc.
  The capitalization is just the reverse the way we normally do.
- In the **knowledge base**, variables are universally quantified. But in **queries**, they are existentially quantified.

  So, `look_alike(X, X).` in a clause it means it is true for all `X`, while in a querry it asks for one particular `X` to make it true:

```
| ?- assertz(look_alike(X, X)).
yes
| ?- look_alike(isaac, isaac).
yes
| ?- look_alike(isaac, X).
X = isaac
yes
```

- Note also the variables are **local** to a single query/clause.

Unification

In the example of the last page, **unification** is done to to **bind** `X` to `isaac`.

It happens automatically when proving predicates requires finding **matched clauses in the KB**: the clause within the query

`look_alike(isaac, X)`

is unified against each clause in the KB defining the `look_alike` procedure, each time with variables renamed:

`look_alike(X1, X1).`

The unifier is {`X1/isaac, X/isaac`}, so `X` is bounded to `isaac`.

Unification can also be requested using the `=` operator. E.g.,

`test(A, B) = test(test(x), test(A))`

binds `A` to `test(x)` and `B` to `test(test(x))`.
In both cases, occurrence checks are omitted.

Some amazing examples

Suppose we have our `parent` procedure, and we also have:

```
sex(cindy, female).
sex(fred, male).
sex(mary, female).
```

We can deduce the child relation…

```
child(X, Y):- parent(Y, X).
```

father, mother, son and daughter relations…

```
father(X, Y):- parent(X, Y), sex(X, male).
mother(X, Y):- parent(X, Y), sex(X, female).
son(X, Y):- child(X, Y), sex(X, male).
daughter(X, Y):- child(X, Y), sex(X, female).
```

and grandparent!

```
grandparent(X, Z):- parent(X, Y), parent(Y, Z).
```

Note that we introduce a variable **within** the body. Anyway, it is interpreted like "for all `X`, `Y` and `Z` the implication holds.

Descriptive programming

We can do all of them in C or C++ as well. But the difference is…

- In Prolog, one "usually" just **describe** the problem and the Prolog engine does the rest! I.e., we didn't think about how our program runs.
- E.g., without taking any care the following works…
  Assuming we also have `parent(grace,fred)` and `sex(grace,female)`

```
| ?- father(fred, cindy).  %% Is Fred Cindy's father?
yes
| ?- son(fred, X).         %% Fred is son of whom?
X = grace
yes
| ?- daughter(X, fred).    %% Who is Fred's daughter?
X = cindy
yes
| ?- grandparent(X, Y).    %% Any grandparent relation you know?
X = grace
Y = cindy
yes
```


Bounded and unbounded variables

- At any time, a Prolog variable can either be **bounded** or **unbounded**. A bounded variable is one which has a **known** (perhaps partial) value, while an unbounded variable is one which value is **not assigned yet**.
- The built-in `var` and `nonvar` predicate can be used to test whether a variable is unbounded or bounded.

```
| ?- look_alike(Y, Z), var(Y), var(Z).
Z=Y
yes
| ?- look_alike(X, isaac), var(X).
no
| ?- look_alike(X, happy(Y)), var(X).
no
```

- They allow functions to **do differently things** depending on whether `X` is **input** (i.e., bounded) or **output** (i.e., unbounded).

"Anonymous" variables

- There is an **anonymous** variable, written like "_".
- Its effect: **a variable name that does not appear** (and thus, cannot be used) anywhere else.
- You'll be surprised by the frequency that you need it… E.g., if you want to get the an element in a `date` structure:

```
get_date_year(date(Y, _, _), Y).
get_date_month(date(_, M, _), M).
get_date_day(date(_, _, D), D).
```

  In programs, the byte-compiler warns if you write `get_date_day(date(Y, M, D)` with `M` and `D` unused.

  so that `get_date_day(date(73,11,2), D)` will bind `D` to `2`.
  The 2nd variable is an "output variable". You will get used to "return value is an argument to be binded" once you do any real work with Prolog.

Operators

- We can express **arithematic expression** with a structure. E.g., `2/3+3** -(4*5)` can be expressed like this…

```
'+'('/'(2, 3), '**'(3, '-'('*'(4, 5))))
```

  While functors have to be atoms, they need not contain only letters!

- It can express anything, but not easy to read. So some functors are defined as "operators". So the above two forms are actually **equivalent**, and prefer the former or output. E.g.,

```
| ?- X = '+'('/'(2, 3), '**'(3, '-'('*'(4, 5)))).
X = 2/3+3** - (4*5)
yes
```

- There are precedence rules to make the above combines "sensibly".
- "`,`", "`;`", "`:-`" and "`->`" are also operators, meaning "and", "or", "head-body" and "if-then". So all clauses are structures.
  So we can `assertz((happy(isaac):-happy(fiona);holiday(today)))`, with extra parentheses for overriding precedence.

Arithematic in Prolog

- We have seen in the last slide that unification won't cause structures to be evaluated as arithematic expression, which is usually a **good** thing.
- If you have a structure in form of an arithematic expression, you can **evaluate it using the "is"** construct:

```
| ?- X is 2 + 5 * (3 + 7).   %% Bind X to 52
```

- The evalutaion is **one-way**: Prolog isn't a equation solver.

```
| ?- 3 is 2 + X.   %% error
```

- Comparisons, like >, <, >=, =<, =:= (equals) and =\= (not equals), are **predicates**, which also evaluate arguments as arithematic expressions:

```
| ?- 3 < 2 + 5. %% yes
| ?- 3 =\= 3. %% no
| ?- X =:= 3. %% error
```

Comparisons of atoms

- We can lexigraphically compare terms without evaluation, using the (strange looking) operators ==, \==, @<, @=<, @> and @>=:

```
| ?- abc == abc. %% yes
| ?- abc(2) == abc(2). %% yes
| ?- 3 \== a. %% no
| ?- 9 @< 2 + 5. %% yes: numbers are smaller than structures
| ?- 3 \== X. %% yes: unbounded variables differs from values
| ?- X=3, 3 \== X. %% no.  Now X is bounded.
```

- Note that things can be different at the beginning, and the same at the end. So use them very carefully.

```
| ?- 3 \== X, X=3, 3 == X.
```

Initially 3 is not the same as X, but after binding it is, so the whole thing is true.

Lists

Structures can also be used to build a **list**. The special functor . is used for that purpose. The empty list is a special element:

`[]`

A list with one element (say, 10) looks like this:

`'.'(10, [])`

A list with two elements (say, 10 and 20) looks like this:

`'.'(10, '.'(20, []))`

A list with two elements, one being a list (say `[10]`) and the other is not (say, 20) looks like this:

`'.'('.'(10, []), '.'(20, []))`

This is all easy… except that it's hard to read.

Syntactic sugar for Lists

- To avoid having to write cryptic lists, Prolog prefers using the following notation **on input and output**:

  `[10],[10, 20],[[10], 20]`

- We can also make list by adding elements at the **front** of a list, e.g.,

```
| ?- X=[1,2,3], Y=[0, a | X]. %% Y=[0, a, 1, 2, 3]
```

- This notation also **work on unification**. E.g.,

```
| ?- [X|Y] = [1,2,3].
X = 1
Y = [2,3]
yes
```

- It is possible to have "improper" list, i.e., something that looks like a list but does not end `[]`. This is displayed like `[1,2|3]`.
  I.e., `'.'(1, '.'(2, 3))`.

Something left?

- **Where is functions**, and how to express that two "things" are **equal**?
  They are in FOL as we know about…
- Does structure work? Not quite… `birthday(isaac)` is a particular object, and it will never be "equivalent" to `date(73,11,2)`…
- The answer: **Prolog does not support them**.
- In other words, if two terms are syntactically the same, they are the same. Otherwise they are different.
- This makes Prolog much easier to implement, but we now have to think twice before using **Skolemization**.
  As the Skolem constant will not be "equal" to anything else.

More complicated example

A procedure that counts the number of elements in a list:

```
count([], 0).
count([_|X], N):- count(X, N1), N is 1 + N1.
```

Note that the return value is in a variable to be bounded. To use it, we can make a query like this:

```
| ?- count([1,2,3], X).  %% X is bounded to 3
```

We can even do the reverse, e.g., generate a list of 3 elements

```
| ?- count(X, 3). %% X is bounded to [_,_,_]
```

Or even more complicated scenarios:

```
| ?- X = [_, a|_], count(X, 4). %% X is bounded to [_,a,_,_]
```

But… **why it works??**

How it works

- Suppose we ask `count(X,N)`. We get the **first answer** from the first clause, `count([], 0)`. This gives the answer `X=[], N=0`.
- It also matches the second clause, with `X=[_|Xa], N=N1a`. So we can **continue** the search on the second clause, which **recursively** ask about `count(Xa, Na)`.
- Of course, this will give us the same solutions for `Xa` and `Na`—starting from `Xa=[], Na=0`.
- For each such solution, `A` is just prepending a "don't care" variable to to `Xa`, while `N=N1a` is just adding one to `Na`. So second solution we get is `X=[_], N=1`.
- If we get this as second solution, `Xa, Na` must also get `Xa=[_], Na=1` as second solution. So our third solution would be `X=[_,_], N=2`.
- The argument continue for all possible `N`…

The need for better control

- If we ask for count(X,Y), it is natural that we will **get infinitely many answers**, one per positive integer for Y.
- But what if we ask for **count(X,4)**? It will give us the first solution `[_,_,_,_]`, and after that…
- If we ask for alternatives, it tries very hard to get more solution.
  … until all the memory is used up. Then the whole Prolog interpreter is aborted.
- Can we modify `count` so that it **stops** instead?
- **Basic chaining** can't do it. But in Prolog, we can **override some chaining steps**. The extra effort might or might not worth it, depending on your application.

Continuations

When a goal have multiple answers, they are returned to us **one-by-one**. A **continuation** is used to keep track of where we left off at.

When we are executing an…

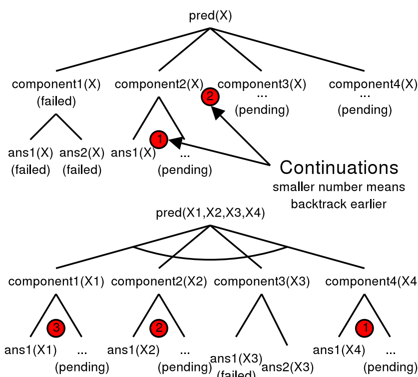
- **OR** clause: at a time, only one component is asked for answers, so only need to be a continuation describing **which component** we are currently asking, and how far we are in that component.
- **AND** clause: each time we require one answer from each of the components, although all of them can have multiple answers. So we need one continuation point for **each** of the components. Whenever something failed, we **backtrack** to the nearest component.

In both cases, the continuation will be omitted if all the possibilities are already tried.

In a picture

OR nodes

AND nodes



Controlling backtracking: cut

- **The mechanism**: there is a predicate, cut (`!`), which always succeed, and, as a side-effect, **remove all continuations currently present** for the currently execution of the running procedure as a side-effect.
- Example: if we have the following…

```
p(1).
p(2):- !.
p(X):- !, p(X).
p(3).
```

  Then p(1) will give you infinitely many "yes" answer, p(2) will give you exactly 1 "yes" answer, p(X) will give you exactly 2 bindings (X=1 and X=2), and p(3) will loop infinitely.
- Why? E.g., for `p(X)`, all clauses matches. But after the first two clauses are examined, the continuation is removed. Lacking a continuation, `p` stops.

How cut in one procedure affects the others?

- Suppose a procedure `q` uses another procedure `p`. The called `p` cuts some continuation. What will happen to `q`?
- **Answer**: it will continue **normally**. In particular, **continuations** of `q` will **not** be removed.
- In general, whenever you call **another** procedure, you are insulated from its cuts.
- So the procedure takes some bounded and unbounded values as input, and produces 0 or more answers at the end (if no error occurs). **You don't need to know anything about its internal**.
  Just like in C++, you don't need to know the internals of a function to call it.

A wrong use of cut

- E.g., we might say that we want to **remove all continuation point once we successfully found a good binding to N**. Our procedure becomes:

```
count([], 0).
count([_|X], N):- count(X, N1), N is 1 + N1, !.
```

- But this **won't work**… It works if we ask `count(X,1)` or `count(X,2)`, but now `count(X,3)` fails! Why?
- `count(X,3)` calls `count(Xa,N1a)`. What `count(Xa,N1a)` return?
- Answer: only **two** bindings, `{Xa/[],N1a/0}` and `{Xa/[_],N1a/1}`. Once it find the latter answer, the cut removed its continuation!
  So the one we need, `{Xa/[_,_],N1a/2}`, is never returned.
- The rationale: adding cuts require **very careful analysis** not to affect the original correct meaning.
  I.e., we can no longer do "descriptive programming".

How to do it right?

So let's ask the real question: **when is it safe to cut** in count?

- Suppose we found a good binding for N1. We can be sure that there is no other solution if N is 1+N1 **and N is originally bounded**.
- Recall that `nonvar` tests **whether a variable is bounded**. Thus our condition for cut is `nonvar(N), N is 1+N1, !`
  We must not reverse them: `N is 1+N1, nonvar(N)` won't work, as the former `is` predicate causes `N` to be non-variable.
- How to incorporate it into our rule? We either have the above, or the normal path. So we can use "or" (semicolon):

```
count([], 0).
count([_|X], N):- count(X, N1),
  ( nonvar(N), N is 1+N1, !
  ; N is 1+N1 ).
```


Guarding against bad values

But is it good enough?

- Let's give it a negative number and ask for lists: `count(X, -1)`.
- It **loops forever**. Our cut didn't stop it: "N is 1+N1" never unify.
- Is it possible to do a "preliminary test"? In particular, **if N is negative, don't bother to try it.** We can use cut to stop it trying, but after cut we want to answer fail rather than a binding.
- There is a "fail" clause which always fail. Thus we can do it this way:

```
count(_, N):- nonvar(N), N < 0, !, fail.
count([], 0).
count([_|X], N):- count(X, N1),
  ( nonvar(N), N is 1+N1, !
  ; N is 1+N1 ).
```


Efficiency problem…

There is one more problem: efficiency . This is a **very slow** `count`: quadratic time. It takes 2 seconds to generate a 1000-element list.

Why it is slow? E.g., when we ask `count(X,10)`, it tries `count(Xa,Na)`, which have to **discard 8 values** before we get N=9.

Each of the discarded values comes from another call of `count(Xb,Nb)`, which, **similarly** have to **discard** many value and call `count` recursively.

It is now not surprising that the procedure is slow. If we want efficiency ,we have to improve it further. See `count_v3.pl`.

The idea: avoid discard and backtrack. Simplified version:

```
count(L, N):- count_internal(L, N, 0).
count_internal([], N, N).
count_internal([_|L], N, Acc):-
        NewAcc is Acc + 1, count_internal(L, N, NewAcc).
```

Note: the third argument of `count_internal` is always bound.

But we don't really need that…

… because the standard predicate `length` does exactly what we have done. Other useful list predicates:

- append(X1, X2, X3): true if concatenating X1 and X2 give you X3. E.g.,

```
| ?- append([a], [b], X).            %% Bind X to [a,b]
| ?- append(X, [d,e], [a,b,c,d,e]). %% Bind X to [a,b,c]
```

- reverse(X1, X2): true if X1 and X2 are exact reverse of each other. E.g.,

```
| ?- reverse([a,b], X).            %% Bind X to [b,a]
| ?- reverse([_], [d,e]).          %% fail
```

- nth(N, List, Element): true if the N-th element of List is Element. E.g.,

```
| ?- nth(2, [a,b], X).            %% Bind X to b
| ?- nth(2, X, b).                %% Bind X to [_,b|_]
```

- member(Term, List): true if Term is within List.

A few more interesting tricks

- `unify_with_occurs_check/2`: do unification with occurrence check. E.g., `unify_with_occurs_check(f(X),X)` fails rather than loops.
- `findall/3`: finds all solutions for a query and put them into a list. E.g., `findall(X, happy(X), L)` might give `L=[isaac, may]`.
- `\+/1`: try call its argument, and return yes if and only if it fails. So `\+ X=1` fails, `\+ 2 is 2 + 1`. It is "approximately" negation.
- `once/1`: try call its argument, but remove all continuations after the first answer is found. This is a "local" version of cut.

And there are many more… see the man page.

Doing procedural things

Many things that can be done by conventional programming languages can be done in Prolog as well. E.g., Assign and use a global variable:

```
set_var:- g_read(a, X), Y is X * X, g_assign(sqr_a, Y).
```

Repeat something 5 times:

```
work5:-for(_, 1, 5), write('Hello, world'), fail; true.
```

Repeat it forever:

```
work_forever:-repeat, write('Hello, world'), fail.
```

Repeat it until some condition holds:

```
work_til_stop:-g_assign(a, 1), repeat, (
    write('Hello, world\n'),
    g_read(a, X), Y is X * 2, g_assign(a, Y),
    Y > 8, !).
```