

Why C++ is not good enough for our task

Lecture 12

Prolog

Now we know what are systems that keep knowledge. But how to make them into concrete programs?

Using a conventional programming language is not a good choice. We will see one popular language in this regard: Prolog. Expect to have to adjust your understanding about what are programming language a little bit.

Reference:

- Textbook: Section 10.3, pp304–306.
- GNU Prolog documentation: within our department server (e.g., virtue),
`/usr/local/GNU/gprolog-1.2.1/gprolog-1.2.1/doc/manual.ps`

AI(0270)

Can we write a reasoning program that does **forward chaining and backward chaining using C++**? Definitely yes, 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.

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.

It would be nice if we can **combine the goods of these choices**.

AI(0270)-12.1

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) => Parent(x, y)
Father(x, y) => 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, June)*. If it output anything, report.
 3. Call *Father(x, June)*. 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.
 5. There must be a way to **continue a search** (i.e., back-track) to get alternative answers.

AI(0270)-12.2

Prolog: overview

- It would be better if the function is automatically generated.
- Prolog does exactly this. We **won't write any C++ functions**. Instead, a Prolog program consists solely of the "rules", i.e., clauses.
- All clauses related to each predicate becomes one predicate function.
- The function returns a binding, and also a "**continuation**" which can be used to continue the search.
- Whenever a binding **fails** subsequently, the continuation automatically used to find **alternative bindings**.
- **Unification** is directly supported, although the **occurrence check** is left out. So sometimes Prolog needs forever to unify without success.
In practice this does really matter that much.
- Only **depth first, backward chaining** is supported.

AI(0270)-12.3

Overview, continued

- **Functions are not supported**. Instead, it provides **structures**, which allows objects to be **composed by others**.
- The result of lacking functions is that **syntactically different objects are assumed to be different**.
- **List and numbers** are directly supported objects. Arithmetic evaluations can also be done, although it lacks the ability to solve even the most simple equation.
- Some predicates have **side effect** like outputting a string on the screen. So Prolog is a **real programming language**.
Rather than just an inference system that deals with rules and facts.
- To support this, usually **backtracking needs to be controlled** in some ways.
E.g., we don't want the same message to be printed many times when we back-track past the clause that print the message.

AI(0270)-12.4

Simple objects

In Prolog, a simple object is an atom, an integer, or a real number.

- 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.
```

You can even write escape sequence within quotes like other languages, 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
```

AI(0270)-12.5

Prolog variables

- A variable has the same rule as **atoms without quotes**, except that it begins with a **capital letter**. E.g.,

`X, B2, First_Queen, etc.`

- No declaration** is needed. Like our CNF, variables are universally quantified. E.g., if we write in our program

`good(X).`

Then `good(1), good('abc/d')`, etc., are all true.

- At any time, a Prolog variable can either be **bounded** or **unbounded**. A bounded variable is one which has a **known value**, while an unbounded variable is one which value is **not assigned yet**.
- Variables are bounded to a value by **unification**. The variable value can only change during **backtracking**.

AI(0270)-12.6

Unification; the Prolog Interpreter

Unification is done by the = operator. E.g.:

| ?- X = a.

X = a

yes

| ?- X = Y.

Y = X

yes

| ?- a = b.

no

| ?-

- This is done in a **Prolog interpreter**, which reads Prolog queries from input and answer them.

- Each query has an answer, which is either yes (meaning successful) or no (meaning failed).

- If the answer is yes and there are variables in the query, the interpreter **gives one of the binding**, and ask whether you want more.

- Sometimes the Prolog reasoning system knows that there is no more solutions, and hence won't ask you about more solution.

AI(0270)-12.7

More about running programs in GNU prolog

- In GNU Prolog, only queries can be interactively typed. **Programs**, on the other hand, must be recorded in a file, compiled and loaded.
- Compiling and loading can be done in an interpreter, using "[]". Faster code can be compiled and executed non-interactively.
- E.g., if the file test.pl contains a line "good(X).":

```
>gprolog
...
| ?- [test].
compiling ../test.pl for byte code...

yes
| ?- good(10).

yes
| ?-
```

AI(0270)-12.8

Predicates

- Prolog predicates are written in the form we are familiar with: **a predicate symbol followed by a pair of parentheses enclosing objects**.

- The predicate name **must be a valid atom name**. Usually we use lower case letters completely.

- Example:

```
mother(june, jane).
mother(bob, jane).
mother(kathy, jane).
%% same predicate symbol can work differently
%% given different number of objects
mother(jane).
```

- Try this: with the above program, ask `mother(X, jane)` in the interpreter. Type ? to see the available options.

- Note also that % is the comment character.

AI(0270)-12.9

Clauses in Prolog

- All sentences, or **clauses**, of Prolog ends with a period sign.
- We have seen the first form of Prolog clauses: **a simple predicate**.
- But of course we need a way to specify **horn sentences** in the form:

$$P(x) \wedge Q(x) \wedge R(x) \Rightarrow S(x)$$

- In Prolog, it is written like: `s(X) :- p(X), q(X), r(X).`
It seems like that in Prolog we need to write everything just the opposite way as we normally would do.
- We can also have **or** on the right hand side, e.g.,
`holiday(X) :- saturday(X); sunday(X).`
Just like writing two clauses `holiday(X):-saturday(X)` and `holiday(X):-sunday(X)`.
- These are backward chaining rules, and predicates are tried from left to right.

AI(0270)-12.10

A full example

Our earlier example of the guilty West is represented like this:

```
criminal(X) :- american(X), weapon(Y), nation(Z),
             hostile(Z), sells(X, Z, Y).
hostile(nono).
nation(nono).
missile(m1).
own(nono, m1).
sells(west, nono, X) :- missile(X), own(nono, X).
american(west).
weapon(X) :- missile(X).
```

A query looks like `weapon(m1).` or `criminal(X).`

Note that **variables are local to a clause**: the X in `sells` and the X in `weapon` are completely unrelated. Each time the procedure is executed, a new X variable is instantiated.

AI(0270)-12.11

Dynamic rules

It is not very useful if all rules **must be written into the program**. So Prolog provides a way for **additional clause to be added** dynamically.

- The procedure must be **marked as dynamic in the program** to prepare for dynamic rule management. E.g.,


```
:- dynamic(hostile/1, nation/1).
```

 This is the syntax used for "declarations" of programs, i.e., things that are done when the program is read or compiled, not when the program executes.
- Then you can use `asserta` (add at front) `assertz` (add at end) to add new knowledge into it when the program runs. E.g.,


```
| ?- assertz(hostile(nono)).
```
- One can also remove facts using `retract`. E.g.,


```
| ?- retract(hostile(nono)).
```

AI(0270)-12.12

Compound object 1: Structures

- Special characters like `/`, `*`, `+`, etc. can be used to build a structures from multiple simple objects. E.g.,

```
2/11/93, 2+3*4
```

- There is also a more "functional form", which looks pretty like a predicate. E.g.,

```
point(2,3), date(2, nov, 1973)
```

- Both of them accepts variables in between, and do unifications. E.g.,

```
| ?- date(X, Y, 1973) = date(Z, nov, 1973).
```

```
Y = nov
Z = X
```

```
yes
```

AI(0270)-12.13

Compound object 2: Lists

- Prolog also **directly support list**. It is written using the notation we know: square brackets surrounding elements. E.g.,


```
| ?- X=[1, one, 3, two, 5, three].
```
- One can make list by adding elements to the front of another list, e.g.,


```
| ?- X=[1,2,3], Y=[0, zero | X].
```
- We can also have list within:


```
| ?- X=[1/1, [nosmell, nostench],
          1/2, [nosmell, stench]].
```
- We can also have variables within list, e.g.,


```
| ?- [X, Y, Z] = [5, X, [2,P]].
```
- Unifications are correctly done. E.g.,


```
| ?- [X | Z] = [1, 2, 3, 4]. % X = 1, Z = [2, 3, 4]
```

AI(0270)-12.14

Arithmetic in Prolog

- If you have a structure in form of an arithmetic expression, you can **evaluate it using the "is" construct**:

```
| ?- X is 2 + 5 * (3 + 7).
```

- The unification is one-way: the other way will fail:

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

- The **unification operator doesn't do evaluation!** E.g.,

```
| ?- X = 2 + 5. %% X is now the structure 2+5.
| ?- 2 + X = 2 + (3 + 4). %% X = 3+4
| ?- 2 + X = 2 + 3 + 4. %% fail: 2+3+4 means (2+3)+4
| ?- 3 + X = 2 + 5. %% fail
```

- We also have comparison operators like `>`, `<`, `>=`, `<=`, `=`, `=:=` (equals) and `:=` (not equals) which does arithmetic evaluations:

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

AI(0270)-12.15

Comparisons of atoms

So what if we want to lexicographically compare two terms without evaluation?

- We can do it with the (strange looking) comparison 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. %% yes, X bounded to 3.
```

AI(0270)-12.16

Don't care value

- There is a special Prolog variable called the **don't care** variable.
- The variable can take any value, so it unify with anything. The effect is the same as a **variable name that does not appear anywhere else**.

- For example, our previous `good(X)` example can also be written like

```
good(_).
```

In fact, you would like to do this because the compiler will warn you if you write `good(X)`, saying that the variable `X` is not used elsewhere.

- This is more useful when you have many rules in a single predicate:

```
invalid_date(_/D/_):- D < 0; D > 31.
invalid_date(M/_/_):- M < 0; M > 12.
invalid_date(M/31/_):- M = 4; M = 6; M = 9; M = 11.
invalid_date(2/D/_):- D > 29.
invalid_date(2/D/Y):- Z is Y mod 4, Z := 0, D > 28.
```

AI(0270)-12.17

More complicated example

A predicate 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:

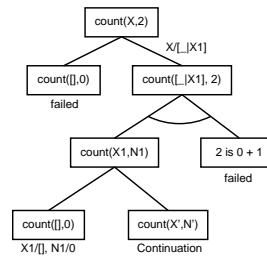
```
| ?- X = [_ , a|_], count(X, 4). %% X is bounded to [_ , a, _ , _]
```

But... **why it works??**

AI(0270)-12.18

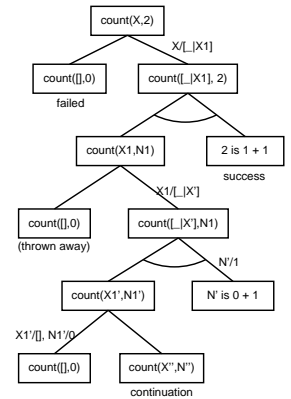
How it works

Suppose we ask count(X,2).



NB: branches with no arc means "or", branch with arc means "and".

Use continuation to retry.



AI(0270)-12.19

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 alternative solution, it will **loop forever** trying to get more solution.
In fact, it won't loop forever. Instead, it will loop until all the memory allocated for continuation points is used up. Then the whole Prolog interpreter terminates.
- Is it possible to write a count procedure which is **more clever**?
- This is the **deficiency of the chaining algorithm**: we won't know to stop. There is no way to solve the problem in general.
- We can **override some chaining steps**. The extra effort might or might not be worth it, depending on your application.

AI(0270)-12.20

Controlling backtracking: cut

- **The idea**: if we are at certain states, we want to **remove all continuation points** of a procedure. The "!" clause (called a cut) does exactly this.
- E.g., we might say that we want to **remove all continuation point once we successfully found a good binding to N**. Our predicate 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!
- It is quite clear from our picture in p.12.18 that count(X,3) will fail: even if we found a good binding to N, we **might need to backtrack** because the caller does not like our binding.
- The rationale: adding cuts require **very careful analysis**.
It is natural: we override part of a full system, so we have to do it in a sane way.

AI(0270)-12.21

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**.
- There is a predicate (nonvar) which **succeed if and only if a symbol is bounded** (and we have "var" which does the reverse thing).
- So our condition for cut is nonvar(N), N is 1+N1, !
- 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 ).
```

- Note that we can use () to group clauses together.

AI(0270)-12.22

Guarding against bad values

But is it good enough?

- Let's try giving 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 tells you that the clause is not true. 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 ).
```

AI(0270)-12.23

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.

AI(0270)-12.24

Let's do the 8-Queen!

- Now let's do something more fancy: **ask for a solution of the 8 queen problem**, given the description.

- So what is a solution for the 8 queen problem? Let's represent it as a list of 8 integers. So a **solution** to the 8 queen problem is a **list of 8 elements, with the first 8 lines being good**:

```
queen(Config):- length(Config, 8), good(Config, 8).
```

- What do we mean by "the first n lines being good"? It means no conflicting queen. The first 0 line is always good.

```
good(_, 0).
```

- For other lines, we need to tell whether the assignment to the list element is a good one. So we have the `valid_number` predicate:

```
valid_number(X):-
    X = 1; X = 2; X = 3; X = 4;
    X = 5; X = 6; X = 7; X = 8.
```

AI(0270)-12.25

8-Queen problem: continue

Now how to know that a configuration is good if size is larger than 1?

- For the first N lines of a configuration Conf to be good: the first N-1 lines must be good.
- And then the N-th line have a valid value.
- And the value must not conflict with the N-1 previous lines. So

```
good(Config, N):-
    N > 0,
    N1 is N - 1,
    good(Config, N1),
    nth(N, Config, Choice),
    valid_number(Choice),
    no_conflict(Config, N, N1).
```

AI(0270)-12.26

8-Queen problem: continue 2

Our last problem: how we know there is no conflict with first N1 lines?

- No conflict if N1 is 0.

```
no_conflict(_,_,0).
```

- Otherwise, no conflict if the first N1-1 lines has no conflict, and the N-th element and the N1-th element is not in the same column or diagonal.

```
no_conflict(Config, N, N1):-
    N1 > 0,
    N2 is N1 - 1,
    no_conflict(Config, N, N2),
    nth(N, Config, P),
    nth(N1, Config, P1),
    P =\= P1,
    P+N =\= P1+N1,
    P-N =\= P1-N1.
```

AI(0270)-12.27