Faculty of Computer Science, Dalhousie University *28/29-Nov-2023*

CSCI 4152/6509 — Natural Language Processing

Lab 10: Prolog Tutorial 2

Lab Instructor: Sigma Jahan and Mayank Anand
Location: Goldberg CS 134(u)/CS 143(g)
Notes author: Vlado Keselj and Phil Cox

Prolog Tutorial 2

Prepared with contributions by Phil Cox (bicycle example).

Lab Overview

- This is the second part of the Prolog tutorial
- We will first cover a more complex example of a bicycle knowledge-base modeling, which illustrates how Prolog can capture semantics
- The rest of the examples show use of Prolog in parsing natural languages, including parsing with DCG grammars, and PCFG in Prolog

Files to be submitted:

1. `parse.prolog`
2. `dcg.prolog`
3. `dcg-ptree.prolog`
4. `dcg-agr.prolog`
5. `dcg-pcfg.prolog`
6. `dcg-agr2.prolog`

Step 1. Logging in to server timberlea

As before, our first step is to login to the server `timberlea` and prepare the appropriate directory.

• Login to the server `timberlea`
As in the previous labs, login to your account on the server `timberlea`.

• Change directory to `csci4152` or `csci6509`
Change your directory to `csci4152` or `csci6509`, whichever is your registered course. This directory should have been already created in one of your previous labs.

• `mkdir lab10`
• `cd lab10`
Now, using the command '`mkdir lab10`' create the directory `lab10`. After this, you should make this directory your current directory using the command: '`cd lab10`'.

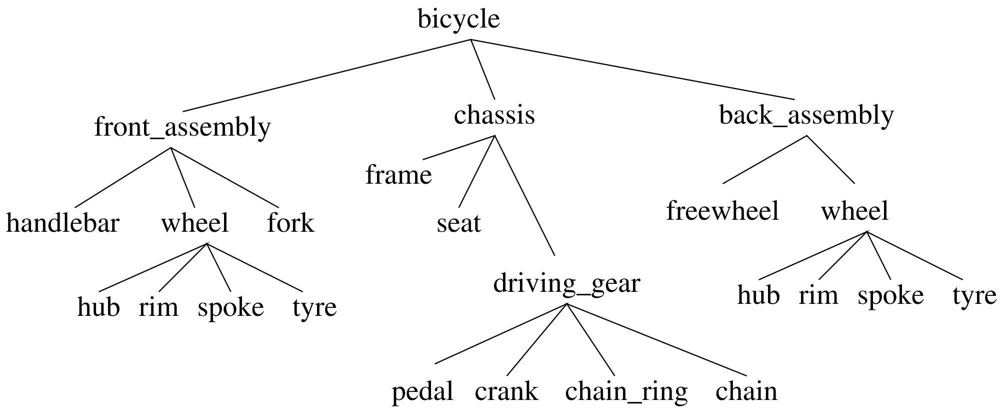
A Review of Basic Elements of Prolog Programs

- Constants; e.g.: `1.2`, `a`, `'string'`
- Variables; e.g.: `Long_name`, `X`, `Y123`, `X_Y`

- _ (underscore) is a special, anonymous variable
- Term expression or functional expression:
  - expression of the form $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms.

Step 2: Bicycle Example

- Let us consider the following hierarchy of bicycle parts:



Bicycle Parts Database in Prolog

```
%
% Predicate: bpart
%
bpart(bicycle). bpart(front_assembly). bpart(handlebar).
bpart(wheel). bpart(hub). bpart(rim). bpart(spoke).
bpart(tyre). bpart(fork). bpart(chassis). bpart(frame).
bpart(seat). bpart(driving_gear). bpart(pedal).
bpart(crank). bpart(chain_ring). bpart(chain).
bpart(back_assembly). bpart(freewheel). bpart(wheel).
bpart(hub). bpart(rim). bpart(spoke). bpart(tyre).
```

- Also available on `timberlea` in directory:
`~prof6509/public`
as file `bpart.prolog`

Prolog line comments. The percent character (`%`) is used in Prolog to mark line comments, as we can notice in the previous example. In other words, the part of line starting with `%` to the end is ignored by the Prolog interpreter. As we would expect, an exception to this rule is any percent character enclosed in a string.

Listing Parts

- Save the file `bpart.prolog` or copy it
- Run Prolog and load the file:

```
swipl
['bpart.prolog'].
```

- Now you can run several query examples:

```
?- bpart(fork).
?- bpart(roof).
?- bpart(X).
```

– Remember to type semicolon (`;`) after each answer in the last query to list all answers.
– Exit Prolog (`halt.`) and prepare another file

Compiling Prolog. As you can notice, working only in one terminal with Prolog with entering and exiting an editor and then entering and exiting the Prolog interpreter repeatedly is not the most efficient way to work. A better way is to open two terminals, and keep an editor open in one and the interpreter in another terminal. A Prolog file can be compiled and loaded in the interpreter window using the command '`['file'].`' as we saw before. Another standard command to load a file in Prolog is:
`consult('file').`
After changing a file in the editor, instead of loading the file again, a better way is to use the SWI Prolog command:
`make.`

To understand why 'make' is better than loading the file again, we can note that 'load' always adds the facts and rules from a file to the current Prolog environment. For example, if we remove some facts from the file, this will not be reflected in the environment after loading the file again. However, 'make' will properly update the predicates.

Direct Part Relations

– Edit or copy the file `part.prolog` (from the same directory)

```
%
% Predicate: part
%
part(bicycle,front_assembly).
part(bicycle,chassis).
part(bicycle,back_assembly).

part(front_assembly, handlebar).
part(front_assembly, wheel).
part(front_assembly, fork).

part(wheel, hub).
part(wheel, rim).
part(wheel, spoke).
part(wheel, tyre).    ...and so on
```

The complete file:

```
%
% Predicate: part
%
part(bicycle,front_assembly).
part(bicycle,chassis).
part(bicycle,back_assembly).

part(front_assembly, handlebar).
part(front_assembly, wheel).
part(front_assembly, fork).
part(wheel, hub).
part(wheel, rim).
part(wheel, spoke).
part(wheel, tyre).

part(chassis, frame).
part(chassis, seat).
part(chassis, driving_gear).

part(driving_gear,pedal).
part(driving_gear,crank).
part(driving_gear,chain_ring).
part(driving_gear,chain).

part(basic_assembly, freewheel).
part(basic_assembly, wheel).

part(wheel,hub).
part(wheel,rim).
part(wheel,spoke).
part(wheel,tyre).
```

Predicate: component

– Finally, edit or copy `component.prolog`:

```
%
% Predicate: component
%
component(X,X) :- bpart(X).
component(X,Y) :- part(X,Z), component(Z,Y).
```

– After loading these files, you can try queries:

```
?- part(bicycle, chassis).
?- part(bicycle,hub).
?- part(bicycle,X).
?- part(X,bicycle).
?- part(X,Y).
?- component(X,fork).
?- component(chassis,X).
```

This example shows how Prolog can be used to model and store a knowledge base in certain domain. In addition to describing all bicycle parts as a list of symbols, we also introduce a binary predicate, i.e., a relation, ‘`part`’ between parts, and then a more complex relation ‘`component`’. We then saw how we can query this knowledge base.

Now, we will see how we can use Prolog to parse a natural language using a simple English example.

Step 3: Using Prolog to Parse NL

Example: Let us consider a simple CFG to parse the following two sentences: “the dog runs” and “the dogs run”

The grammar is:

```
S  -> NP VP        N  -> dog
NP -> D N          N  -> dogs
D  -> the          VP -> run
                   VP -> runs
```

Difference Lists

Difference list is a way of representing a list as a difference between two lists, e.g. the list

```
[the,dog]
```

can be represented as a difference of the following pairs of lists

```
[the,dog],[]
[the,dog,runs,home],[runs,home]
[the,dog,runs],[runs]
[the,dog|R],R
```

Using Difference Lists

The problem of parsing using this grammar can be expressed in the following way in Prolog:

```
s(S,R)  :- np(S,I), vp(I, R).
np(S,R) :- d(S,I), n(I,R).
d([the|R], R).
n([dog|R], R).
n([dogs|R], R).
vp([run|R], R).
vp([runs|R], R).
```

Save this in file `parse.prolog`. On Prolog prompt we type:

```
?- ['parse.prolog'].
% parse.prolog compiled 0.00 sec, 1,888 bytes
true.
?- s([the,dog,runs],[]).
true.
?- s([runs,the,dog],[]).
false.
```

Submit parse.prolog

– Submit the file `parse.prolog` using the `nlp-submit` command.

Step 4: Definite Clause Grammars (DCG)

Type this example in file `dcg.prolog`:

```
s2  --> np, vp.
np --> d, n.
```

```
d  --> [the].
n  --> [dog].
n  --> [dogs].
vp --> [run].
vp --> [runs].
```

DCG rules get translated into Prolog rules with difference lists.

Type in the Prolog interpreter:

```
?- ['dcg.prolog'].
...
?- s2([the,dog,runs],[]).
...
?- s2([runs,the,dog],[]).
...
```

Submit the file `dcg.prolog` using the command `nlp-submit`.

Step 5: Building a Parse Tree

DCG rules can contain arguments.

A parse tree can be built in the following way:

```
s(s(Tn,Tv))    --> np(Tn), vp(Tv).
np(np(Td,Tn)) --> d(Td), n(Tn).
d(d(the))      --> [the].
n(n(dog))      --> [dog].
n(n(dogs))     --> [dogs].
vp(vp(run))    --> [run].
vp(vp(runs))   --> [runs].
```

Save this program as file: `dcg-ptree.prolog`

In Prolog Interpreter:

At Prolog prompt, after we load the file, we type the query and obtain a result as follows:

```
?- s(X, [the, dog, runs], []).
 X = s(np(d(the),n(dog)),vp(runs)).
```

Submit the file `dcg-ptree.prolog` using the command `nlp-submit`.

Step 6: Handling Agreement

Prepare the following program in file: `dcg-agr.prolog`

```
s(s(Tn,Tv))       --> np(Tn,A), vp(Tv,A).
np(np(Td,Tn),A)  --> d(Td), n(Tn,A).
d(d(the))         --> [the].
```

```
n(n(dog),sg)      --> [dog].
n(n(dogs),pl)     --> [dogs].
vp(vp(run),pl)    --> [run].
vp(vp(runs),sg)   --> [runs].
```

This grammar will accept sentences “the dog runs” and “the dogs run” but not “the dog run” and “the dogs runs”. Other phenomena can be modeled in a similar fashion.

Prolog Interpreter

Try parsing the following sentences in Prolog interpreter:
the dogs run
the dog run
the dogs runs
the dog runs

Submit the file `dcg-agr.prolog` using the command `nlp-submit`.

Step 7: PCFG in Prolog

Embedded Code

We can embed additional Prolog code using braces, e.g.:

```
s(T)   --> np(Tn), vp(Tv), {T = s(Tn,Tv)}.
```

and so on, is another way of building the parse tree.

Expressing PCFGs in Prolog

We will now look at how a PCFG can be expressed using DCG-style code. PCFGs (Probabilistic Context-Free Grammars) are covered (or will be covered soon) in class in more detail in lectures.

Let us consider the following example of a PCFG:

S	→	NP VP	/1	VP	→	V NP	/.5	N	→	time	/.5
NP	→	N	/.4	VP	→	V PP	/.5	N	→	arrow	/.3
NP	→	N N	/.2	PP	→	P NP	/1	N	→	flies	/.2
NP	→	D N	/.4					D	→	an	/1
V	→	like	/.3					P	→	like	/1
V	→	flies	/.7								

The probabilities can be passed as an additional argument, and calculated using embedded code:

```
s(T,P) --> np(T1,P1), vp(T2,P2),
           {T = s(T1,T2), P is P1 * P2 * 1}.
np(T,P) --> n(T1,P1), {T = n(T1), P is P1 * 0.4}.
```

and so on. A full DCG code for the above PCFG could be:

```
s(s(Tn,Tv),P) --> np(Tn,P1), vp(Tv,P2), {P is P1 * P2}.
np(np(T),P) --> n(T,P1), {P is P1 * 0.4}.
np(np(T1,T2),P) --> n(T1,P1), n(T2,P2), {P is P1 * P2 * 0.2}.
np(np(Td,Tn),P) --> d(Td,P1), n(Tn,P2), {P is P1 * P2 * 0.4}.
v(v(like), 0.3) --> [like].
v(v(flies), 0.7) --> [flies].
p(p(like), 1.0) --> [like].
vp(vp(Tv,Tn), P) --> v(Tv, P1), np(Tn, P2), {P is P1 * P2 * 0.5}.
vp(vp(Tv,Tp), P) --> v(Tv, P1), pp(Tp, P2), {P is P1 * P2 * 0.5}.
pp(pp(Tp,Tn), P) --> p(Tp, P1), np(Tn, P2), {P is P1 * P2}.
n(n(time), 0.5) --> [time].
n(n(arrow), 0.3) --> [arrow].
n(n(flies), 0.2) --> [flies].
d(d(an), 1.0) --> [an].
```

Write this Prolog program into the file `dcg-pcfg.prolog`, load this grammar into the Prolog interpreter, and then after issuing the query:
`?- s(T,P,[time,flies,like,an,arrow],[]).`
the interpreter would reply with:

```
T = s(np(n(time)), vp(v(flies), pp(p(like), np(d(an), n(arrow)))))
P = 0.0084
```

and after typing `;` (semi-colon), we get:

```
T = s(np(n(time), n(flies)), vp(v(like), np(d(an), n(arrow))))
P = 0.00036
```

After typing second '`;`', the interpreter reports 'false' since there are no more parse trees.

Submit: `dcg-pcfg.prolog`

– Submit the file `dcg-pcfg.prolog` using the command `nlp-submit`

Step 8: An Extended Example

- Start with a copy of the previous example:

```
cp dcg-agr.prolog dcg-agr2.prolog
```

- Let us implement a rule for '-s' inflection

You may have noticed that we needed separate rules for 'dog' and 'dogs', and similarly for 'run' and 'runs', to encode noun and verb inflections. We will assume that these are regular in our grammar, and we will capture them by two Prolog rules:

- Remove the rules:

```
n(n(dogs),pl)     --> [dogs].
vp(vp(runs),sg)   --> [runs].
```

- and add the following rules:

```
n(n(Npl),pl) --> [Npl],
  { atom_concat(Nsg, 's', Npl), n(_,sg,[Nsg],[]) }.
vp(vp(Vsg),sg) --> [Vsg],
  { atom_concat(Vpl, 's', Vsg), vp(_,pl,[Vpl],[]) }.
```

In the above rules, we use the Prolog predicate `atom_concat`, which can be used to make concatenation of two atom names into a new atom. For example, predicate `atom_concat(a,b,X)` would lead to the variable `X` getting the value `ab`. However, the predicate is implemented in such way that it can be used to remove prefix or suffix. For example, `atom_concat(X,s,dogs).` would give `X = dog`, or `atom_concat(dog,X,dogs)` would give `X = s`. Interestingly, we can use also use `atom_concat(X,Y,dogs)` to get all splits of the atom 'dogs' into two substrings.

In the Prolog Interpreter

- Try new grammar in the interpreter:

```
?- s(T,[the,dog,runs],[]).
```

- You should obtain a proper parse tree
- Remember to type semicolon (`;`) if you do not get a prompt after answer
- Try also sentences 'the dogs run', 'the dog run', and 'the dogs runs'
- We could now add more words, for example:

```
n(n(dog),sg) --> [dog].
n(n(cat),sg) --> [cat].
```

- However, there is a more compact way to do this:
- Remove rules:

```
n(n(dog),sg) --> [dog].
vp(vp(run),pl) --> [run].
```

Using a Word List

- Add the following rules:

```
n(n(X),sg) --> [X], { member(X, [dog, cat]) }.
vp(vp(X),pl) --> [X], { member(X, [run, walk]) }.
```

- Try parsing sentences 'the dog runs', 'the cat runs', 'the dogs walk', 'the cat walks' and similar
- The predicate 'member' is predefined predicate in SWI-Prolog, but in case that it is not and you get an error, you can define it by adding the following two rules:

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

- Add the nouns 'turtle' and 'rabbit' and VPs 'swim' and 'crawl' to the grammar
- Try parsing more sentences

Submit: `dcg-agr2.prolog`

– Submit the file `dcg-agr2.prolog` using the command `nlp-submit`

End of the Lab.