Practical Session 8: Prolog

1 Introduction: strings

Before diving in the main aspect of this session (DCG), we must first introduce how strings work in Prolog. As you might guess, strings are represented as lists. Specifically, lists of ASCII codes. But as typing ascii codes by hand can be tedious, Prolog provides double quotes as syntactical sugar to represent such lists. The string_codes predicate can help you go from one form to the other (see also the documentation for **name** and similar conversion predicates):

```
?- string_codes("hello", L).
L = [104, 101, 108, 108, 111].

?- string_codes(S, [119, 111, 114, 108, 100]).
S = "world".
```

Given a list of codes, you can also use format to print it as a string directly:

```
?- A = [104, 101, 108, 108, 111],
   B = [119, 111, 114, 108, 100],
   format("~s ~s~n", [A, B]).
```

2 DCG Basics

This session will explore a feature of Prolog called DCG. Definite Clause Grammars allow you to define a language grammar directly into Prolog, to able to generate or parse sentences from this language. It is merely a layer on top of regular Prolog, and the DCG rules can be directly translated into normal Prolog.

DGC is very similar to BNF. A DCG rule has the form:

```
head --> body.
```

and is analogous to a standard Prolog rule for:

```
head :- body.
```

The body can contain terminals (a prolog list) or non-terminals (mostly other DCG rules). For instance, to define a language accepting sentences that are series of the letter a:

```
as --> [].
as --> [a], as.
```

Use the phrase(as, P) predicate to generate some examples of sentences from this language and to check whether one particular sentence is part of this language or not.
Then modify the toy example to accept sentences made of alternating a's and b's :

```
P = [] ;
P = [a] ;
P = [a, b] ;
P = [a, b, a] ;
P = [a, b, a, b] ;
P = [a, b, a, b, a] ;
```

etc.

A terminal can also (and perhaps more usefully) be a string.

```
as --> [].
as --> "a", a.

?- phrase(as2, P),format('~s~n', [P]).
```

It is possible to specify alternatives with ;:

```
article_phrase --> ("a" ; "an"), "␣", noun.

noun --> "book".
noun --> "car".
```

Try the above grammar in the shell.

3 Prolog and DCG

Using arguments to rules can allow you to capture and manipulate the input string. For instance:

```
digit(D) --> [D], {code_type(D, digit)}.
```

The { ... } construct allows you to run a regular Prolog predicate from within a DCG rule. In this case, we use the code_type predicate to check that D is a digit (refer to the documentation for more details on code_type).

Write rules to accept simple Scheme expressions to define real constants. It should accept:

```
        (define foo 12345)
        (define bar2 3.1415).
        (define baz 00000145.149800)
        (define zork .4242)
        (define blob 4242.)
```

But not:

```
        (define glob "hello")
        (define glub 12.345.41)
```

Check with phrase that your grammar accepts or refuses those sentences.

We can define a simple grammar for well-formed boolean algebra formulas as the following:

```
<b-expression> ::= <b-term> [OR <b-expression>]
<b-term>       ::= <not-factor> [AND <b-term>]
<not-factor>   ::= [NOT] <b-factor>
<b-factor>     ::= <b-literal> | <b-variable> | (<b-expression>)
<b-literal>    ::= 0 | 1
<b-variable>   ::= <identifier>
```

Write DCG rules to accept formulas in this grammar.

4 Parse trees

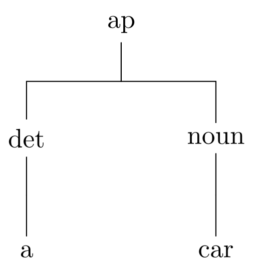
When parsing a sentence, DCG rules need to figure out the parse tree of this sentence, but this tree it is not given explicitly. However is it easy to obtain this tree from the rules, using arguments to match the parsed structure and build the tree using unification. Consider for instance:

```
article_phrase(ap(Det, Noun)) --> det(Det), "␣", noun(Noun).

det(det(an))  --> "an".
det(det(a)) --> "a".

noun(noun(book)) --> "book".
noun(noun(car)) --> "car".
```

If you parse the sentence "a car" you would obtain: ap(det(a), noun(car)) which would correspond to the tree:



Modify your rules for boolean formulas to produce a parse tree.

5 Interpreting the parse tree

Once you have a parse tree, you can feed it to an interpreter which will perform whatever actions the parse tree represents.

Write an eval predicate which evaluates a boolean formula. Boolean variables should be in lowercase and treated as Prolog facts. For instance:

```
?- assert(a).
?- assert(b:- false).
```

Should then return true if you evaluate the tree for "a OR b" but obviously false if you evaluate "a AND b".