Practical Session 3: Prolog 3

Strings

SWI Prolog has two kind of strings: `` `hello` `` and `"hello"`. The first one is just syntactic sugar for a list of character codes. You can type `print(`hello`)` at the prompt to see this. Double-quoted strings use a more efficient representation, but you can convert between the two formats by using the `string_codes/2` predicate.

Difference Lists

A difference list is a way to represent a list as the "difference" between two lists. Namely, the second list is to be substracted from the end of the first list. For instance, you can represent the list `[a, b]` as the difference between `[a, b, c, d]` and `[c, d]`.

Things start getting really interesting when we use unbound variables in difference lists. For instance, we can represent `[a, b]` as the difference between `[a, b | T]` and `T`. Consider the following rule:

```
append(I/M, M/O, I/O).
```

Here, `/` is used as a simple functor. Now run the following query:

```
?- append([a, b | T1] / T1,  [c, d | T2] / T2,  R).
T1 = [c, d | T2],
R  = [a, b, c, d | T2].
```

We have just managed to append two list in O(1) instead of O(n) like we did during the first lab. The downside is that you can only pull this trick once (because you can only bind the variable once).

Definite Clause Grammars (DCG)

A DCG is a way to represent a grammar (for instance, defining the syntax of a

programming language) as Prolog rules. Because DCGs are Prolog, they are in fact much more powerful than this: you can use them to go from an abstract syntax tree to the program text, or to enumerate valid programs.

More pragmatically, DCG are a tool to match lists. Consider the following trivial DCG (this is valid Prolog):

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

This grammar matches lists comprised of zero or more atoms `a`. You can use `phrase/2` to perform a match, or list all valid inputs:

```
?- phrase(alphas, [a, a, a]).
true.

?- phrase(alphas, [a, a, b]).
false.

?- phrase(alphas, L).
L = [];
L = [a];
L = [a, a] ...
```

There is also `phrase/3`, which allows us to match only a prefix of the input:

```
phrase(alphas, [a, a, c], T).
T = [a, a, c] ;
T = [a, c] ;
T = [c] ;
false.
```

The matched input is now represented as a *difference list* between the supplied input (`[a, a, c]`) and `T`. In fact, DCGs are all about difference lists:

```
?- listing(alphas).
alphas(A, A).
alphas([a|A], B) :-
        alphas(A, B).
```

DCG clauses are translated into regular prolog clauses with two additional parameters, representing a difference list. Above, the first clause binds the tail end

of the difference list to the rest of the input. The second clause ensures that the list begins with an `a`, then recursively calls itself on its tail, passing the tail end of the difference list along.

By the way, the tail end of the difference list (the `B` parameter) corresponds to the third parameter of `phrase/3`.

It is illuminating to see the listing for a DCG clause where multiple clauses are chained together:

```
alpha --> [a].
beta  --> [b].
gamma --> [c].
chain --> alpha, beta, gamma.

?- listing(chain).
chain(A, D) :-
        alpha(A, B),
        beta(B, C),
        gamma(C, D).
```

The tail end of the difference list represents the "rest" of the input: what the clause didn't consume. You can see in the listing how the rest of the input from one clause becomes the input for the clause that follows it.

Advanced DCGs

What is valid inside a DCG clause? In our `alphas` example we had reference to DCG clauses (in this case, to `alphas` itself), and matching elements of the list by enclosing them in a list (`[a]`). In practice, many other things are possible:

- Matching more than one elements: `[a, b, c]`.
- Executing regular Prolog clauses:

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

- DCG clauses with parameters:

  ```
  alphas(R, R) --> [].
  alphas(C, R) --> { C1 is C+1 }, [a], alphas(C1, R).
  ```

```
?- phrase(alphas(0, C), [a, a, a]).
C = 3.
```

- Matching strings (this shouldn't be a surprise, as strings are just lists): try replacing `[a]` with `` `a` `` in `alphas`. More surprisingly, `"a"` works as well (for legacy reasons).

Remember that if you have any doubts about the semantics, you can always use `listing/1` to see how things are translated to regular Prolog.

Exercises

Exercise `[grammar]`

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>
```

And identifiers are single lowercase letters (use `code_type/2`).

Write DCG rules to accept the formulas defined by this grammar.

Exercise `[tree]`

The DCG you wrote in the previous exercise simply recognizes whether the input is a valid formula. Modify it so that it also constructs an AST (Abstract Syntax Tree).

For instance, the tree for the formula `a AND NOT b OR 1` should look something like `or(and(a, not(b)), 1)`. Use `atom_string/2` for string <> atom conversion.

Exercise `[interpreter]`

Now that you have an AST, write an interpeter for it: `interpret(AST, Value)`.

Write a `truth/1` predicate to statically assign values to logical variables. For instance:

```
truth(a, 1).
truth(b, 0).

?- interpret(and(a, not(b)), X).
X = 1.
```

The bitwise arithmetic operators might be useful (`apropos("arithmetic functions")`).

Exercise `[regex]`

Implement a parser and an interpreter for basic regular expressions, as described below (from Implementing Regular Expressions (https://swtch.com/~rsc/regexp/)):

> Regular expressions are a notation for describing sets of character strings. When a particular string is in the set described by a regular expression, we often say that the regular expression matches the string.
>
> The simplest regular expression is a single literal character. Except for the special metacharacters `*+?()|` , characters match themselves. To match a metacharacter, escape it with a backslash: `\+` matches a literal plus character.
>
> Two regular expressions can be alternated or concatenated to form a new regular expression: if `e1` matches `s` and `e2` matches `t` , then `e1|e2` matches `s` or `t` , and `e1e2` matches `st` .
>
> The metacharacters `*` , `+` , and `?` are repetition operators: `e1*` matches a sequence of zero or more (possibly different) strings, each of which match `e1` ; `e1+` matches one or more; `e1?` matches zero or one.
>
> The operator precedence, from weakest to strongest binding, is first alternation, then concatenation, and finally the repetition operators. Explicit parentheses can be used to force different meanings, just as in arithmetic expressions. Some examples: `ab|cd` is equivalent to `(ab)|(cd)` ; `ab*` is equivalent to `a(b*)` .

If you're really brave, you can even attempt to implement your interpreter using a

finite automata, as described in the above link.
