USER GUIDE TO EXTENDED DCG LIBRARY FOR TUPROLOG

The library is encapsulated in the 2p.jar file. Open the application and insert the following predicate into your theory:

```
load_library('alice.tuprolog.lib.EDCGLibrary').
```

The purpose of this guide is to introduce the users to the new tools added to the old DCG Library. You can use predicates phrase/2 and phrase/3 as described in the 2p guide available on the official tuProlog website. Please refer to it for the basics about DCG.

With the Extended DCG Library you can now utilize a more compact and convenient notation to write your grammars by using the Extended BNF syntax. Here's a list of the new operators that have been introduced:

Operator	EBNF Syntax	Description
;	X --> A;B	X produces A or otherwise B
*	X --> *A	X produces zero or more occurrences of A
+	X --> +A	X produces one or more occurrences of A
?	X --> ?A	X produces zero or one occurrence of A
^	X --> ^(A,N)	X produces N occurrences of A

So, how can we use those operators efficiently to define our grammars? First, we'll take a look at how to write production rules for a simple parser by adopting an EBNF type of notation, and then we'll show how to add the AST generation.

PARSING

-Multiple Occurences:

Let's consider this very simple example of a grammar considering multiple occurences of the right side of a production rule:

```
x --> *a.
a --> [A], {atom(A)}.
```

x produces zero or more occurences of a, which can be any literal term A, which means that the following sentences are the ones that are going to be successfully parsed:

```
[]. [a]. [a,a]. [a,a,a]. [a,a,a,a].  [...].  [a,b]. [a,b,a]. [a,b,a,b]. etc.
```

It is NOT the same as writing directly:

```
x --> *[A], {atom(A)}.
```

because this type of production rule is not true if A is different kinds of literals, for A can be bound to just one literal. It is also not true for an empty input list, which means that '[]' is NOT a sentence for the described grammar. You need to specify the exception as follows:

```
x --> *[A], {atom(A);A=[]}.
```

[]. [a]. [a,a]. [a,a,a]. [a,a,a,a]. [...]. etc. Those are sentences of the above grammar.
[a,b]. [a,b,a]. [a,b,a,b]. etc. Those are NOT sentences of the above grammar.

So be careful in the way you define the terminal symbols of your grammar.

-Optional occurence:

You can then represent optional choice by using the '?' operator as follows:

```
x --> ?a.
```

which will correctly parse none or a single occurence of 'a', which means either the empty list [] or one occurence of the nonterminal symbol 'a' and it's relative production rule.

-Alternative occurence:

For a more compact definition of your code, you can represent alternative choices in the same production rule by using the ';' operator, as shown by the example below:

```
x --> a;b.
```

This grammar rule is verified if the sentence parsed is an occurence of the 'a' symbol, or otherwise an occurence of the 'b' symbol.

-Multiple occurences with power notation:

The power notation allows you to express your production rules in a really convenient way. You can not only specify exactly how many occurences of a given nonterminal symbol, but even mixing Prolog with DCG to add even more constraints to your rule. Here's an example:

```
x -->  ^(a,3).
```

This production rule indicates that there are exactly 3 occurences of the 'a' symbol. By adding some Prolog code between '{}' you improve the expressivness. For example, you can insert constraints for the number of occurences as follows:

```
x -->  ^(a,N), {N < 4}.
```

As you can see, the possibilities for parsing by using this new notation are almost endless.

PARSING WITH AST GENERATION

Parsing with AST Generation for multiple occurences is a bit more complex, since you have to specify the type of terms to visualize as you abstract syntax. Let's first consider a simple example of AST generation for single productions:

```
x(single(T)) --> a(T).
a(A) --> [A], {atom(A)}.
```

The goal ?-phrase(x(T),[b]). will return the solution in the desired form: T / single(b), instead of just T / b. Basically, all nonterminal symbols are just Prolog terms, and parsing unifies them.
You need to pay attention though when you are dealing with multiple occurences of a nonterminal symbol in a single production rule. Let's look at the next example:

```
x(multi(T)) --> *a(T).
a(A) --> [A], {atom(A)}.
```

Writing our grammar this way is going to return us just one result, which means that by executing goal ?-phrase(x(T),[b,b,b]), the solution is just going to be T / single(b), which is not what we really want!
So, how do we solve the problem? Well, the EDCGLibrary provides just the right tools for this task. To obtain the desired result, which is T / [multi(b),multi(b),multi(b)], we are going to write our production rule for the nonterminal symbol 'x' as follows:

```
x(T) --> *(a(F),single(F),T).
```

It means that 'x' produces a list T of zero or more occurences of 'a' in the specified form, which is basically the same concept as the one of the Prolog predicate 'findall/3'. It's the same deal for the other operator representing multiple productions, and for the power notation, even though the operator we are going to use is not '^' but '#' instead. And remember that you can add Prolog code for more expressiveness. For example,

```
x(T) --> #(num(A),N1,out(A),T1), #(num(B),N2,out(B),T2), {N3 is N1+N2},
         #(num(C),N3,out(C),T3), {append(T1,T2,TT), append (TT,T3,T)}.
num(N) --> [N], {number(N)}.
```

An example of a sentence of this grammar could be [1,1,1,2,2,3,3,3,3,3], which the AST generator returns as [out(1),out(1),out(1),out(2),out(2),out(3),out(3),out(3),out(3),out(3)].
At last, you can set an optional production to generate the corresponding result if the production to parse is contained in the input list, else you can return another result. Here's an example:

```
x(M) --> ?(n(N),N,null,M).
n(N) --> [N], {number(N)}.
parse(L,T):-phrase(x(T),L).
```

So, if you try to parse a token L=[1] for example, it will return T/1. If you parse an empty list, it will return T/null instead. This way, you can have a desired generation in the AST in either cases.
Alternatively, you can simply parse ?n(M) by adding appropriate Prolog code:

```
x(H) --> ?n(M), {(ground(M), H=M); H=null}.
n(N) --> [N], {number(N)}.
```