Grammar Rules in Prolog

Based on Chapter 9 in Clocksin & Mellish

One popular grammar notation is BNF (Backus-Naur Form)

BNF is commonly used in the definition of programming languages

A grammar comprises production rules

A Simple Grammar in BNF

\[ <s> ::= a \ b \]
\[ <s> ::= a \ s \ b \]

- S is enclosed by < > which indicates s is a non-terminal
- A and b are terminal symbols
- Terminal symbols can never be rewritten

Generating Sentences

- A grammar can be used to generate a string of symbols called a sentence
- Start with a non-terminal
- Make substitutions using production rules
- Terminate when the current sequence doesn't contain any non-terminal symbols
Generating Sentences - 2

<s> ::= a b
<s> ::= a <s> b

- Our grammar (given above) can generate sentences (strings) of what form???

Generating Sentences - 3

<s> ::= a b
<s> ::= a <s> b

- Our grammar (given above) can generate sentences (strings) of what form???

Answer: \(a^n b^n\) \(n = 1, 2, 3, \ldots\)

- The set of sentences generated by the grammar is called the language defined by the grammar

Another Example Grammar

- A robot arm can be sent sequences of commands:
  - up: move 1 step upward
  - down: move 1 step downward

- What is a grammar to capture the robot's possible movements??

Another Example Grammar - 2

- A robot arm can be sent sequences of commands:
  - up: move 1 step upward
  - down: move 1 step downward

- What is a grammar to capture the robot's possible movements??

\(<\text{move}> ::= <\text{step}>\)
\(<\text{move}> ::= <\text{step}> <\text{move}>\)
\(<\text{step}> ::= \text{up}\)
\(<\text{step}> ::= \text{down}\)
A grammar generates sentences
A grammar can also be used to recognize a given sentence
Recognition is really the opposite of generation (i.e., determine whether a sentence is part of a language as opposed to generating a sentence that is part of a language)
Sometimes it is called parsing

In Prolog it is easy to write a parsing program using special grammar rule notation called DCG (definite clause grammar)
Most Prolog implementations support this special notation for grammars
Very easy to change BNF to DCG

Starting with our simple example in BNF
\[
<s> ::= a \ b \\
<s> ::= a \ <s> \ b
\]
Converting it to DCG
\[
s --> [a], [b]. \\
s --> [a], s, [b].
\]
From BNF to DCG - 3

- Starting with our robot example in BNF
  \[ \text{< move > ::= < step >} \]
  \[ \text{< move > ::= < step > < move >} \]
  \[ \text{< step > ::= up} \]
  \[ \text{< step > ::= down} \]

- Converting it to DCG

From BNF to DCG - 4

- Starting with our robot example in BNF
  \[ \text{< move > ::= < step >} \]
  \[ \text{< move > ::= < step > < move >} \]
  \[ \text{< step > ::= up} \]
  \[ \text{< step > ::= down} \]

- Converting it to DCG

  \[ \text{move --> step.} \]
  \[ \text{move --> step, move.} \]
  \[ \text{step --> [ up ].} \]
  \[ \text{step --> [ down ].} \]

DCG Notation

- Each sentence is represented by 2 lists
- Difference lists of terminal symbols
- Can think of first list as the sentence you are parsing and the second list as the part of the sentence that is left-over after the parsing is done

Back to our First Example

\[ \text{s --> [ a ], [ b ].} \]
\[ \text{s --> [ a ], s, [ b ].} \]

?- s( [ a, a, b, b ], [ ] ).
yes
?- s( [ a, b, b ], [ ] ).
no
?- s( [ a, a, b, b, c ], [ c ] ).
yes
?- s( [ a, a, c, b, b ], [ c ] ).
no
How Does Prolog DCGs Work? - 1

- When Prolog consults grammar rules it turns them into normal Prolog clauses
- Uses simple rules to turn grammar rules into clauses depending on whether the symbols on the right-hand side of the rules are
  - All non-terminals
  - A mix of terminals and non-terminals
  - All terminals

How Does Prolog DCGs Work? - 2

- If the DCG rule is
  - n -> n1, n2, ... , nn.

- If all the n1, n2, ... nn are non-terminals then the rule is translated into the clause:
  - n( List1, Rest ) :-
    n1( List1, List2 ),
    n2( List2, List3 ),
    ...
    nn( Listn, Rest ).

Back to our Second Example - 1
move --> step.
move --> step, move.
step --> [ up ].
step --> [ down ].

Back to our Second Example - 2
?- move([ up, up, down ], []). yes
?- move([ up, up, down ], [ down ]). yes
?- move([ up, up, left ], []). yes
?- move([ up, up, left ], [ left ]). yes
?- move([ up, X, up ], []). X = up ;
X = down ;
no
How Does Prolog DCGs Work? - 3

◊ If the DCG rule is
  » n -> n1, [t2], n3, [t4].
  » Where n1 and n3 are non-terminals and t2 and t4 are terminals
◊ Then the DCG rule is translated into the following clause:
  » n( List1, Rest ) :-
    n1( List1, [ t2 | List3 ] ),
    n3( List3, [ t4 | Rest ] ).

How Does Prolog DCGs Work? - 4

◊ If the DCG rule is
  » n -> [ t1 ], [t2].
  » Where t1 and t2 are terminals
◊ Then the DCG rule is translated into the following clause:
  » n( [ t1, t2 | Rest ], Rest ).

Translating Example 1

◊ Translate the following DCG grammar into Prolog clauses:
  s --> [ a ], [ b ].
  s --> [ a ], s, [ b ].

Translating Example 1 - 2

◊ Translate the following DCG grammar into Prolog clauses:
  s --> [ a ], [ b ].
  s --> [ a ], s, [ b ].
  s( [ a, b | Rest ], Rest ).
  s( [ a | List1 ], Rest ) :-
    s( List1, [ b | Rest ] ).
Translating Example 2

Translate the following DCG grammar into Prolog clauses:

\[
\begin{align*}
\text{move} & \rightarrow \text{step}. \\
\text{move} & \rightarrow \text{step}, \text{move}. \\
\text{step} & \rightarrow [ \text{up} ]. \\
\text{step} & \rightarrow [ \text{down} ].
\end{align*}
\]

Translating Example 2 - 2

Translate the following DCG grammar into Prolog clauses:

\[
\begin{align*}
\text{move} & \rightarrow \text{step}. \\
\text{move} & \rightarrow \text{step}, \text{move}. \\
\text{step} & \rightarrow [ \text{up} ]. \\
\text{step} & \rightarrow [ \text{down} ].
\end{align*}
\]

\[
\begin{align*}
\text{move} (\text{List}, \text{Rest}) & :- \\
\text{step} (\text{List}, \text{Rest}). \\
\text{move} (\text{List1}, \text{Rest}) & :- \\
\text{step} (\text{List1}, \text{List2}), \\
\text{move} (\text{List2}, \text{Rest}). \\
\text{step} ([\text{up} | \text{Rest}], \text{Rest}). \\
\text{step} ([\text{down} | \text{Rest}], \text{Rest}).
\end{align*}
\]

A More Interesting Example

More interesting examples of grammars come from programming languages and natural languages.

Here is an example grammar for a simple subset of English:

\[
\begin{align*}
\text{sentence} & \rightarrow \text{noun_phrase}, \text{verb_phrase}. \\
\text{verb_phrase} & \rightarrow \text{verb}, \text{noun_phrase}. \\
\text{noun_phrase} & \rightarrow \text{determiner}, \text{noun}. \\
\text{determiner} & \rightarrow [\text{a}]. \\
\text{determiner} & \rightarrow [\text{the}]. \\
\text{noun} & \rightarrow [\text{cat}]. \\
\text{noun} & \rightarrow [\text{mouse}]. \\
\text{verb} & \rightarrow [\text{scares}]. \\
\text{verb} & \rightarrow [\text{hates}].
\end{align*}
\]

A More Interesting Example - 2

\[
\begin{align*}
\text{sentence} & \rightarrow \text{noun_phrase}, \text{verb_phrase}. \\
\text{verb_phrase} & \rightarrow \text{verb}, \text{noun_phrase}. \\
\text{noun_phrase} & \rightarrow \text{determiner}, \text{noun}. \\
\text{determiner} & \rightarrow [\text{a}]. \\
\text{determiner} & \rightarrow [\text{the}]. \\
\text{noun} & \rightarrow [\text{cat}]. \\
\text{noun} & \rightarrow [\text{mouse}]. \\
\text{verb} & \rightarrow [\text{scares}]. \\
\text{verb} & \rightarrow [\text{hates}].
\end{align*}
\]

Example sentences generated by this grammar are:

\[
\begin{align*}
[\text{the}, \text{cat}, \text{scares}, \text{a}, \text{mouse}] \\
[\text{the}, \text{mouse}, \text{hates}, \text{the}, \text{cat}] \\
[\text{the}, \text{mouse}, \text{scares}, \text{the}, \text{mouse}]
\end{align*}
\]
May want to have extra arguments in our parser apart from the ones dealing with consumption of the input series.

For example, let us look at the problem of agreement between the subject and the verb in a sentence.

We can add nouns and verbs in plural.

The following rules could be added to our grammar:

- noun --> [ cats ]
- noun --> [ mice ]
- verb --> [ scare ]
- verb --> [ hate ]

Now we can generate sentences like:

- [ the, mice, hate, the, cats ]

Unfortunately, it will also generate sentences like:

- [ the, mouse, hate, the, cat ]

The problem lies in the rule:

```
sentence --> noun_phrase, verb_phrase.
```

This states that any noun phrase and verb phrase can be put together to form a sentence.

But in English the noun phrase and verb phrase in a sentence must agree in number.

Both must be singular or both must be plural.

Context dependencies cannot be directly handled by BNF grammars, but they can be handled by DCG grammars by adding arguments to non-terminal symbols.

So, we add Number as an argument and modify our grammar.
Adding Extra Arguments - 5

sentence( Number ) --> noun_phrase( Number ), verb_phrase( Number ).
verb_phrase( Number ) --> verb( Number ), noun_phrase( Number1 ).
noun_phrase( Number ) --> determiner( Number ), noun( Number ).
determiner( singular ) --> [ a ].
determiner( Number ) --> [ the ].
noun( singular ) --> [ cat ].
noun( plural ) --> [ cats ].
noun( singular ) --> [ mouse ].
noun( plural ) --> [ mice ].
verb( singular ) --> [ scares ].
verb( plural ) --> [ scare ].
verb( singular ) --> [ hates ].
verb( plural ) --> [ hate ].

Examples with an Extra Argument

?- sentence( plural, [ the, mice, hate, the, cats ], [ ] ). yes
?- sentence( plural, [ the, mouse, hates, the, cat ], [ ] ). no
?- sentence( Number, [ the, mouse, hates, the, cat ], [ ] ).
Number = singular
?- sentence( singular, [ the, What, hates, the, cat ], [ ] ).
What = cat ;
What = mouse ;
no
?- sentence( plural, [ the, mice, hate, the, cat, a, lot ], [ a, lot ] ).

Parse Trees

◦ The parse tree of a phrase is a tree with the following properties:
  » All the leaves of the tree are labelled by terminal symbols of the grammar
  » All the internal nodes of the tree are labelled by non-terminal symbols; the root of the tree is labelled by the non-terminal that corresponds to the phrase
  » The parent-children relation in the tree is as specified by the rules of the grammar

Adding Extra Arguments - 6

◦ Converting DCG rules with arguments into Prolog clauses
◦ When DCG rules are converted to Prolog clauses, the arguments on non-terminals are simply added to the usual two list arguments, with the two lists coming last.
◦ For example:
sentence( Number ) --> noun_phrase( Number ), verb_phrase( Number ).

Is converted into:
sentence( Number, List1, Rest):-
noun_phrase( Number, List1, List2 ), verb_phrase( Number, List2, Rest ).
Sometimes it is useful to have the parse tree explicitly represented in the program to perform some computation on it. For example to extract the meaning of a sentence.

The parse tree for the noun phrase “the cat” would be represented as

> noun_phrase( determiner( the ), noun( cat ) )

To generate a parse tree, add to each non-terminal the parse tree as an argument.

The parse tree of a noun phrase in our grammar has the form

> noun_phrase( determiner( the ), noun( cat ) )

This represents the tree

```
noun_phrase
  
  determiner
    the

  noun
    cat
```

Adding the parse trees as arguments into our noun phase grammar rule

```
noun_phrase --> determiner, noun.
```

Results in the modified rule

```
noun_phrase( noun_phrase( DetTree, NounTree ) ) -->
  determiner( DetTree ), noun( NounTree ).
```

This rule can be read as

> A noun phrase whose parse tree is noun_phrase ( DetTree, NounTree) consists of:
  > A determiner whose parse tree is DetTree, and
  > A noun whose parse tree is NounTree

 Modified Grammar

```
sentence( Number, sentence( NP, VP ) ) -->
  noun_phrase( Number, NP ), verb_phrase( Number, VP ).

verb_phrase( Number, verb_phrase( Verb, NP ) ) -->
  verb( Number, Verb ), noun_phrase( Number1, NP ).

noun_phrase( Number, noun_phrase( Det, Noun) ) -->
  determiner( Det ), noun( Number, Noun ).

determiner( determiner( the ) ) --> [ the ].
noun( singular, noun( cat ) ) --> [ cat ].
noun( plural, noun( cats ) ) --> [ cats ].

etc. etc.
```
Modified Grammar - 2

- When this grammar is read by Prolog it is automatically translated into a standard Prolog program.
- The first grammar rule:
  sentence( Number, sentence( NP, VP )) -->
      noun_phrase( Number, NP ),
      verb_phrase( Number, VP ).
- Is translated into:
  sentence( Number, sentence( NP, VP), List, Rest ) :-
      noun_phrase( Number, NP, List, Rest0 ),
      verb_phrase( Number, VP, Rest0, Rest ).

Adding Extra Tests

- So far, everything mentioned in the grammar rules has had to do with how the input sequence is consumed
- Every goal in the resulting Prolog clause has been involved with consuming some amount of input
- Sometimes we want to specify goals not of this type
- Any goals enclosed in curly brackets { } are to be left unchanged by the translator

Modified Grammar - Example

?- sentence( Number, ParseTree, [ the, mice, hate, the, cat ], [ ] ).
Number = plural
ParseTree = sentence( noun_phrase( determiner( the ),
      noun( mice ) ), verb_phrase( verb( hate ), noun_phrase
      ( determiner( the ), noun( cat ) ) ) )
?- noun_phrase( plural, ParseTree, [ the, mice, hate, the, cat ], Leftover ).
ParseTree = noun_phrase( determiner( the ), noun( mice ) )
Leftover = [ hate, the, cat ]

Example – Extra Tests

- Consider the robot arm example again:
  move --> step.
  move --> step, move.
  step --> [ up ].
  step --> [ down ].
- Now, we want to define the distance the robot has moved as the difference between the robot’s position before the move and after the move.
- Let each step be 1mm in either the positive or negative direction
- So, the program [ up, up, down, up ] would be equivalent to a distance of 2 mm
To accomplish our goal we have to add the move’s distance as an argument.

To the old grammar:

- move --> step.
- move --> step, move.
- step --> [ up ].
- step --> [ down ].

So, we get a revised grammar:

move( D ) --> step( D ).
move( D ) --> step( D1 ), move( D2 ), { D is D1 + D2 }.
step( 1 ) --> [ up ].
step(-1) --> [ down ].

A Final Example

Let's make our robot arm example more interesting.
Suppose the robot can be in 1 of 2 gears: g1 or g2.
When a step command is received in gear g1 the robot will move by 1 mm up or down. In gear, g2 it will move by 2 mm.
The program for the robot should consist of gear commands, step commands and a stop command (ending the program).

Example programs are:

- stop
  - Returns a distance of 0
- g1 up up stop
  - Returns a distance of 2
- g1 up up g2 down up stop
  - Returns a distance of $1 + 1 + 2 \times (-1 + 1) = 2$
- g1 g1 g2 up up g1 up down up g2 stop
  - Returns a distance of $2 \times (1 + 1) + 1 \cdot 1 + 1 = 5$
Here is our grammar with distance but without gears:

move( D ) --> step( D ).
move( D ) --> step( D1 ), move( D2 ), { D is D1 + D2 }.
step( 1 ) --> [ up ].
step( -1 ) --> [ down ].

We need to extend it with new rules to handle the gears:

prog( 0 ) --> [ stop ].
prog( Dist ) --> gear( _ ), prog( Dist ).
prog( Dist ) --> gear( G ), move( D ), prog( Dist1 ),
{ Dist is G * D + Dist1 }.
gear( 1 ) --> [ g1 ].
gear( 2 ) --> [ g2 ].

Here is our grammar with distance but without gears:

move( D ) --> step( D ).
move( D ) --> step( D1 ), move( D2 ), { D is D1 + D2 }.
step( 1 ) --> [ up ].
step( -1 ) --> [ down ].

Extend it with the following new rules to handle the gears:

prog( 0 ) --> [ stop ].
prog( Dist ) --> gear( _ ), prog( Dist ).
prog( Dist ) --> gear( G ), move( D ), prog( Dist1 ),
{ Dist is G * D + Dist1 }.
gear( 1 ) --> [ g1 ].
gear( 2 ) --> [ g2 ].