Slides to Accompany *Programming Languages and Methodologies*

R. J. Schalkoff

Chapter 5, Part 1: Parsing Using Prolog

Parsing and the Prolog LGN Preprocessor

- There are a number of possible approaches to developing parsers for grammar recognition. These include:
  - The CYK Algorithm
  - The use of ('cc') tools like `Bison`
- Prolog provides a very important and useful preprocessor (the LGN) which greatly facilitates parser development.

Logic Grammar Notation (LGN)

- Entering grammar productions directly in Prolog is facilitated by the use of a preprocessor for Logic Grammar Notation (LGN).
- This is also referred to as the Definite Clause Grammar (DCG).
- This is not part of the Prolog standard, but most Prolog implementations provide it.
- Grammar rules in the LGN/DCG format look like ordinary clauses, but use `-->` for separating the head and tail, rather than `:-`.
- Expanding grammar rules is done by an internal predicate `expand_term`, which adds two additional arguments to each predicate.

- The `expand_term` predicate is normally invoked by the compiler/interpreter as a preprocessing step in the consulting of a file.

Using Grammar Rule Notation

It is assumed that the strings to be parser are represented in Prolog as Prolog lists, i.e.,

$$x = aabac$$

is represented in Prolog as the list

```
[a, a, b, a, c]
```

Recall our ultimate objective is to represent our programs as lists, e.g.,

```
[void, main, lparens, void,
        rparens, lbrace, int, x, semicolon,
                |<more>]
```

First, consider a simple example. Suppose we had a very minimal set of productions:

$$S \to AB$$
$$A \to terma$$
$$B \to termb$$

where $S, A, B \in V_N$ and $terma, termb \in V_T$.

This could be entered *directly* in Prolog LGN as:

```
s --> a, b.
a --> [terma].
b --> [termb].
```

Notice in Prolog the *predicates* are s, a and b. Consider the following translation and application in Prolog:

```
?- ['logicgr1.pro'].
```

```
logicgr1.pro compiled, 0.00 sec, 784 bytes.

Yes
?- listing(s).

s(A, B) :-
        a(A, C),
        b(C, B).

Yes
```

Here we see Prolog has done all the work in converting the notation to actual predicates. What is even more important is that Prolog has built a parser for us. We will adopt a "'consumption-based"' interpretation of the parser operation. For example, the LGN-generated clause:

```
s(A, B) :-
        a(A, C),
        b(C, B).
```

may be interpreted as:

> A is an s with B leftover if A is an a, with C leftover and C is a b with B leftover.

Notice the shared variables and that typically B is the empty list (to require the parser to use all of the input string (list)). Similarly,

```
?- listing(a).


a([terma|A], A).


Yes
?- listing(b).
```

```
b([termb|A], A).

Yes
```

More importantly, we can show that the string "`terma termb`", i.e., the list `[terma, termb]` is $\in L(G)$:

```
?- s(All, []).

All = [terma, termb] ;

No

?- s([terma, termb], []).

Yes
```

Prolog and Grammars: The Summary So Far

productions	in Prolog LGN	translation
$S \rightarrow AB$	`s --> a, b.`	`s(A, B) :- a(A, C), b(C, B).`
$A \rightarrow terma$	`a --> [terma].`	`a([terma|A], A).`

More on Grammar Rule Notation

Suppose we had a slightly more complex[a] set of productions:

$$S \rightarrow AB$$

$$S \rightarrow C$$

$$A \rightarrow C$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$B \rightarrow c$$

$$C \rightarrow d$$

Recall that the logic grammar representation in Prolog (below)

[a] Aside: can you determine $L(G)$?

looks slightly different due to the Prolog requirement that predicates must begin with a lowercase symbol.

This is represented in Prolog as:

```
s --> a, b.
s --> c.
a --> c.
a --> [terma].
b --> [termb].
b --> [termc].
c --> [termd].
```

Now watch what Prolog does:

```
6 ?- ['logicgr2.pro'].
logicgr2.pro compiled, 0.00 sec, 60 bytes.
```

```
Yes
7 ?- listing(s).

s(A, B) :-
        a(A, C),
        b(C, B).
s(A, B) :-
        c(A, B).

Yes
8 ?- listing(a).

a(A, B) :-
        c(A, B).
a([terma|A], A).
```

```
Yes
9 ?- listing(b).

b([termb|A], A).
b([termc|A], A).

Yes
10 ?- listing(c).

c([termd|A], A).

Yes
```

Here we see again Prolog has done all the work in converting the logic grammar notation to actual predicates. As noted, you should be able to generate $L(G)$ for this example. Here are some examples:

```
11 ?- s([a, b], []).
```

```
No
12 ?- s([terma, termb], []).

Yes

14 ?- s([termd], []).

Yes

15 ?- s([termd, termb], []).

Yes
16 ?- s([termb, termb], []).

No
```

Another Translation

Consider:

```
s --> a,[b],c,[d].
```

with translation:

```
?- listing(s).
s(A, E) :-
a(A, B),
B=[b|C],
c(C, D),
D=[d|E].
```

More LGN Production Examples

Consider:

```
/* another set of lgn examples */

s1 --> a,[termb],c,[termd].

s2 --> [terma],b,[termc],d.

s3 --> [terma],[termb],[termc],[termd].

a --> [terma].

b --> [termb].

c --> [termc].

d --> [termd].
```

with associated translations:

```
?- listing(s1).
s1(A, E) :-
a(A, B),
B=[termb|C],
c(C, D),
D=[termd|E].

true.

?- listing(s2).
s2([terma|A], D) :-
b(A, B),
B=[termc|C],
d(C, D).

true.

?- listing(s3).
s3([terma|A], D) :-
A=[termb|B],
B=[termc|C],
```

```
C=[termd|D].

true.
```

Here's a few uses:

```
/* straightforward -- generation */

?- s1(What,[]).
What = [terma, termb, termc, termd].

?- s2(What,[]).
What = [terma, termb, termc, termd].

?- s3(What,[]).
What = [terma, termb, termc, termd].

/* straightforward -- parsing */

?- s1([terma, termb, termc],[]).
false.

?- s1([terma, termb, termc,termd],[]).
true.
```

```
?- s1([termf,termb,termc,termd],[]).
false.


/* relax constraint on empty list as second argument */

?- s1(Rev,_).
Rev = [terma, termb, termc, termd|_G222].

?- s2(Rev,_).
Rev = [terma, termb, termc, termd|_G222].

?- s3(Rev,_).
Rev = [terma, termb, termc, termd|_G222].

/* and in parsing: */

?- s1([terma, termb, termc],_).
false.

?- s1([terma,termb,termc,termd,terme,termf],_).
```

```
true.

?- s1([terma,termb,termc,termd,terme,termf],Leftover).
Leftover = [terme, termf].
```

Adding Variables in the LGN

Suppose we need to add another variable to the automatically translated LGN rule above, e.g., what we we really want after LGN translation is:

```
sentence(N, A, B) :-
          nounphrase(N, A, C),
          verbphrase(N, C, B).
```

Here is how we do it in Prolog's LGN:

```
sentence(N) --> nounphrase(N), verbphrase(N).
```

which translates[a] into:

```
sentence(A, B, C) :-
          nounphrase(A, B, D),
          verbphrase(A, D, C).
```

[a]The reader should verify this.

What's Left?

- Convert `minic` productions into LGN (see text)
- Build a scanner in Prolog (see text)
- Parse using Prolog (see text)