Parsing and grammars

A very simple grammar

A very simple parser

Grammar rule notation

Goals in grammar rules

Parameters in non-terminals

Accumulating an integer

A list recognizer

"Real" compilation

Credit: The first part of this section borrows heavily from chapter 12 in Covington, et al.

A very simple grammar

Here is a grammar for a very simple language:

```
Sentence => Article Noun Verb

Article    => the | a

Noun       => dog | cat | girl | boy

Verb       => ran | talked | slept
```

Here are some sentences in the language:

```
the dog ran

a boy slept

the cat talked
```


A very simple parser

Here is a simple parser for the grammar: (`parser0.pl`)

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the| Left], Left). article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Usage:

```
?- sentence([the,dog,ran]).
Yes

?- sentence([the,dog,boy]).
No

?- sentence(S). % Generates all valid sentences
S = [the, dog, ran] ;
S = [the, dog, talked]
...
```


A very simple parser, continued

For reference:

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the|Left], Left). article([a| Left],  Left).
noun([Noun|Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Note that the heads for `article`, `noun`, and `verb` all have the same form.

Let's look at a clause for `article` and a unification:

```
article([the|Left], Left).

?- article([the,dog,ran], Remaining).
Remaining = [dog, ran]
```

If `Words` begins with `the` or `a`, then `article(Words, Remaining)` succeeds and unifies `Remaining` with the rest of the list. **The key idea: `article`, `noun`, and `verb` each consume an expected word and produce the remaining words.**

A very simple parser, continued

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
```

A query sheds light on how `sentence` operates:

```
?- article(Words, Left0), noun(Left0, Left1),
    verb(Left1, Left2), Left2 = [].

Words = [the, dog, ran]
Left0 = [dog, ran]
Left1 = [ran]
Left2 = []

?- sentence([the,dog,ran]).
Yes
```

Each goal consumes one word. The remainder is then the input for the next goal.

Why is `verb`'s result, `Left2`, unified with the empty list?

A very simple parser, continued

For convenient use, let's add a predicate `s/1` that uses `concat_atom` to split up a string based on spaces:

```
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words).

sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1),
    verb(Left1, []).
```

Usage:

```
?- s('the dog ran').
Yes

?- s('ran the dog').
No
```


Grammar rule notation

Prolog's *grammar rule notation* provides a convenient way to express these stylized rules. Instead of this,

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
article([the| Left], Left).
article([a| Left], Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

we can take advantage of grammar rule notation and say this,

```
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

Note that the literals are specified as singleton lists.

The semicolon is an "or". Alternative: `noun --> [dog]. noun --> [cat]. ...`

Grammar rule notation, continued

```
% cat parser1.pl
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

`listing` can be used to see the clauses generated by Prolog:

```
 ?- listing(sentence).
sentence(A, D) :-  article(A, B), noun(B, C), verb(C, D).

?- listing(article).
article(A, B) :-
        (   'C'(A, a, B)
        ;   'C'(A, the, B)
        ).

?- listing('C').
'C'([A|B], A, B).
```

Note that the predicates generated for `sentence`, `article` and others have an arity of 2.

Grammar rule notation, continued

```
% cat parser1a.pl
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words,[]).

sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].

verb --> [ran]; [talked]; [slept].
verb([Verb|Left], Left) :- verb0(Verb).

verb0(jumped). verb0(ate). verb0(computed).
```

Points to note:

`sentence`, `article`, `verb`, and `noun` are known as *non-terminals*. `dog`, `cat`, `ran`, talked, etc. are known as *terminals*.

The call to `sentence` in `s` now has two terms to match the generated rule.

`verb` has both a grammar rule and an ordinary rule. The latter makes use of `verb0` facts.

Goals in grammar rules

We can insert ordinary goals into grammar rules by enclosing the goal(s) in curly braces. Here is a chatty parser that recognizes the language described by the regular expression `a*`:

```
parse(S) :- atom_chars(S,Chars), string(Chars,[]). % parser6.pl

string --> as.

as --> [a], {writeln('Got an a')}, as.
as --> [], {writeln('out of a\'s')}.
```

Usage:

```
?- parse('aaa').
Got an a
Got an a
Got an a
out of a's
Yes
```

What would the behavior be if the order of the `as` clauses was reversed?

Parameters in non-terminals

We can add parameters to the non-terminals in grammar rules. The following grammar recognizes `a*` and produces the length, too.

```
parse(S, Count) :-                              % parser6a.pl
    atom_chars(S, Chars), string(Count, Chars, []).

string(N) --> as(N).

as(N) --> [a], as(M), {N is M + 1}.
as(0) --> [].
```

Usage:

```
?- parse('aaa',N).
N = 3

?- parse('aaab',N).
No
```


Parameters in non-terminals, continued

Here is a grammar that recognizes $a^N b^{2N} c^{3N}$: (`parser7a.pl`)

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []).

string([N,NN,NNN])
    --> as(N), {NN is 2*N}, bs(NN), {NNN is 3*N}, cs(NNN).

as(N) --> [a], as(M), {N is M+1}. as(0) --> [].

bs(N) --> [b], bs(M), {N is M+1}. bs(0) --> [].

cs(N) --> [c], cs(M), {N is M+1}. cs(0) --> [].

?- parse('aabbbbcccccc', L).
L = [2, 4, 6]
?- parse('abbc', L).
No
```

Can this language be described with a regular expression?

Problem: Recognize $a^X b^Y c^Z$ where `X <= Y <= Z`.

Parameters in non-terminals, continued

A parser for $a^X b^Y c^Z$ where `X <= Y <= Z`: (`parser7b.pl`)

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []).

string([X,Y,Z]) --> as(X), bs(Y), {X =< Y}, cs(Z), {Y =< Z}.

as(N) --> [a], as(M), {N is M+1}. as(0) --> [].
bs(N) --> [b], bs(M), {N is M+1}. bs(0) --> [].
cs(N) --> [c], cs(M), {N is M+1}. cs(0) --> [].
```

Usage:

```
?- parse('abbbcccccc', L).
L = [1, 3, 6]

?- parse('aabbc', L).
No

?- parse('c', L).
L = [0, 0, 1]
```


Accumulating an integer

Here is a parser that recognizes a string of digits and creates an integer from them:

```
parse(S,N) :- % parser8.pl
    atom_chars(S, Chars), intval(N,Chars,[]), integer(N).

intval(N) --> digits(Digits), { atom_number(Digits,N) }.

digits(Digit) --> [Digit], {digit(Digit)}.
digits(Digits) -->
    [Digit], {digit(Digit)},
    digits(More), { concat_atom([Digit,More],Digits)}.

digit('0'). digit('1'). ... digit('9').
```

Usage:

```
?- parse('4312', N).
N = 4312
```


A list recognizer

Here is a parser that recognizes lists consisting of integers and lists: (`list.pl`)

```
parse(S) :- atom_chars(S, Chars), list(Chars, []).

list --> ['['], values, [']'].
list --> ['['], [']'].

values --> value.
values --> value, [','], values.

value --> digits(_).
value --> list.
% assume digits from previous slide

?- parse('[1,20,[30,[[40]],6,7],[]]').
Yes

?- parse('[1,20,,30,[[40]],6,7],[]]').
No

?- parse('[1, 2, 3]').  % Whitespace!  How could we handle it?
No
```


"Real" compilation

These parsing examples fall short of what's done in a compiler. The first phase of compilation is typically to break the input into "tokens". Tokens are things like identifiers, individual parentheses, string literals, etc.

The input

```
[ 1, [30+400], 'abc']
```

might be represented as a stream of tokens with this Prolog list:

```
[lbrack, integer(1), comma, lbrack, integer(30), plus,
integer(400), rbrack, comma, string(abc), rbrack]
```

The second phase of compilation is to parse the stream of tokens and generate code (traditional compilation) or execute it immediately (interpretation).

We could use a pair of Prolog grammars to parse source code. The first one would parse character-by-character and generate a token stream like the list above. The second grammar would parse that token stream.