Grammar Rules

The Parsing Problem

Representing the Parsing Problem in Prolog

The Grammar Rule Notation

Grammar of a Language

A set of rules

for specifying what sequences of words are acceptable as sentences of the language.

Grammar specifies:

How the words must group together to form phrases.

What orderings of those phrases are allowed.

Parsing Problem

Given:

A grammar for a language and a sequence of words

Problem:

Is the sequence an acceptable sentence of the language?

Simple Grammar Rules for English

Structure Rules:

```
sentence    --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.
```

Simple Grammar Rules for English (Ctd.)

Valid Terms:

```
determiner  --> [the].

noun        --> [man].
noun        --> [apple].

verb        --> [eats].
verb        --> [sings].
```

Reading Grammar Rules

`X --> Y`:

"X can take the form Y"

`X,Y`:

"X followed by Y"

`sentence --> noun_phrase, verb_phrase`:

A `sentence` can take a form: `noun_phrase` followed by `verb_phrase`

Alternatives

Two rules for `verb_phrase`:

```
verb_phrase --> verb, noun_phrase.

verb_phrase --> verb.
```

Two possible forms:

`verb_phrase` can contain a `noun_phrase`:
"the man eats the apple"

or it need not:
"the man sings"

Valid Terms

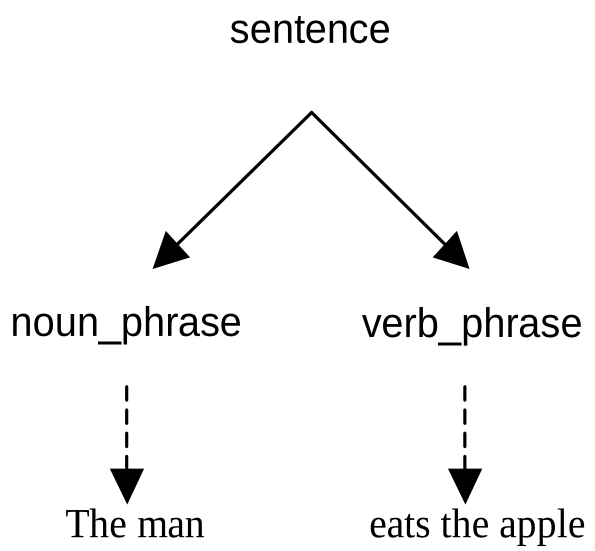
Specify phrases made up in terms of actual words
(not in terms of smaller phrases)

```
determiner --> [the].
```

A `determiner` can take the form: the word `the`.

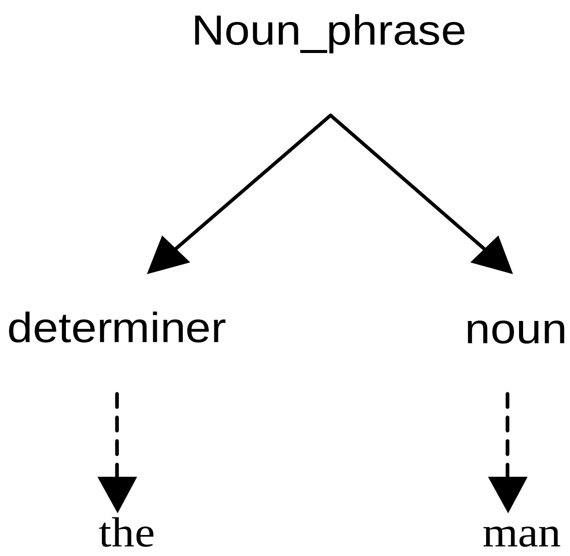
Parsing

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



Parsing

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



How To

How to test whether a sequence is an acceptable sentence?

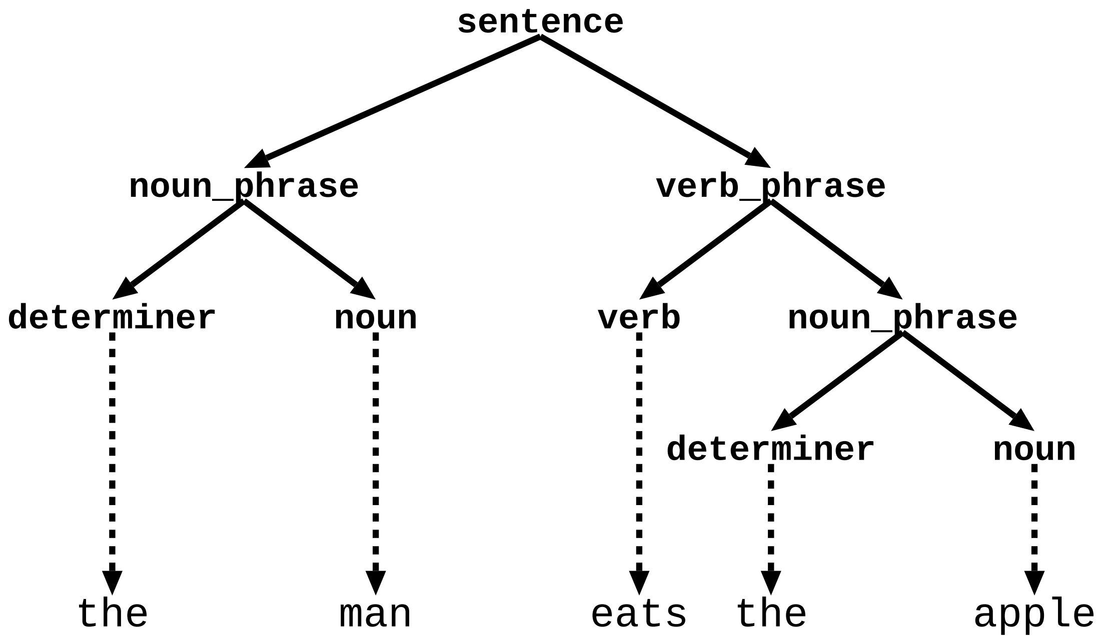
Apply the first rule to ask:
Does the sequence decompose into two phrases,
acceptable `noun_phrase` and acceptable `verb_phrase`?

How to test whether the first phrase is an acceptable `noun_phrase`?

Apply the second rule to ask:
Does it decompose into a `determiner` followed by a `noun`?

And so on

Parse Tree


sentence
noun_phrase
verb_phrase
determiner
noun
verb
noun_phrase
determiner
noun
the
man
eats
the
apple

Parsing Problem

The problem of constructing parse tree for a sentence, given a grammar

Prolog Parse

Problem

Parse a sequence of words

Output

True

This sequence is a valid sentence

Example Representation

Words as Prolog atoms

and

Sequences of words as lists

`[the,man,eats,the,apple]`

Sentence

`sentence(X)`

means

X is a sequence of words forming a grammatical sentence

`sentence([the,man,eats,the,apple])`

yields **True**

`noun_phrase(X)`

X is a noun phrase

`verb_phrase(X)`

X is a verb phrase

Program

```
sentence(X) :-
     append(Y,Z,X),
     noun_phrase(Y), verb_phrase(Z).

verb_phrase(X) :-
     append(Y,Z,X),
     verb(Y), noun_phrase(Z).

verb_phrase(X) :-
     verb(X).

noun_phrase(X) :-
     append(Y,Z,X),
     determiner(Y), noun(Z).
```

```
determiner([the]).
noun([apple]).
noun([man]).
verb([eats]).
verb([sings]).
```

Inefficient

A lot of extra work

Unnecessary Searching

Generate and Test

Generate a sequence

Test to see if it matches

Simplest Formulation of the search
but inefficient

Inefficiency

The program accepts sentence "the man eats the apple":

```
| ?- sentence([the,man,eats,the,apple]).

yes
```

`append(Y,Z,[the,man,eats,the,apple])` on backtracking can generate all possible pairs:

```
Y=[], Z=[the,man,eats,the,apple]
Y=[the], Z=[man,eats,the,apple]
Y=[the,man], Z=[eats,the,apple]
Y=[the,man,eats], Z=[the,apple]
Y=[the,man,eats,the], Z=[apple]
Y=[the,man,eats,the,apple], Z=[]
```

Redefinition

`noun_phrase(X,Y)`

there is a noun phrase at the beginning of the sequence X

and

the part that is left after the noun phrase is Y

```
?- noun_phrase([the,man,saw,the,cat],[saw,the,cat])
```

should succeed

```
noun_phrase(X,Y) :- determiner(X,Z), noun(Z,Y).
```

Improved Program

```
sentence(S0,S) :-
    noun_phrase(S0,S1),
    verb_phrase(S1,S).

noun_phrase(S0,S) :-
    determiner(S0,S1),
    noun(S1,S).

verb_phrase(S0,S) :-
    verb(S0,S).
verb_phrase(S0,S) :-
    verb(S0,S1),
    noun_phrase(S1,S).
```

```
determiner([the|S],S).

noun([man|S],S).
noun([apple|S],S).

verb([eats|S],S).
verb([sings|S],S).
```

Goal

`sentence(S0,S)` means:

There is a sentence at the beginning of `S0`

and

what remains from the sentence in `S0` is `S`

We want whole `S0` to be a sentence

i.e., `S` should be empty

```
?-sentence([the,man,eats,the,apple]),[]).
```

Do you remember difference lists?

Pros and Cons

Advantage: More efficient

Disadvantage: More cumbersome

Improvement idea:

Keep the easy grammar rule notation for the user
Automatically translate into the Prolog code for computation

Grammar Rule Notation

Defining Grammars

Prolog provides an automatic translation facility for grammars

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

translates to:

```
sentence(S0,S) :- noun_phrase(S0,S1), verb_phrase(S1,S).
```

```
determiner --> [the]
```

translates to

```
determiner([the|S],S).
```

Now, the user can input the grammar rules only:

```
sentence    --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

determiner  --> [the].

noun        --> [man].
noun        --> [apple].

verb        --> [eats].
verb        --> [sings].
```

It will be automatically translated into:

```
sentence(S0,S) :-
    noun_phrase(S0,S1),
    verb_phrase(S1,S).

noun_phrase(S0,S) :-
    determiner(S0,S1),
    noun(S1,S).

verb_phrase(S0,S) :-
    verb(S0,S).
verb_phrase(S0,S) :-
    verb(S0,S1),
    noun_phrase(S1,S).
```

```
determiner([the|S],S).

noun([man|S],S).
noun([apple|S],S).

verb([eats|S],S).
verb([sings|S],S).
```

Goals

```
?-sentence([the,man,eats,the,apple],[]).
yes

?-sentence([the,man,eats,the,apple],X).
X=[]
```

SWI-Prolog provides an alternative (for the first goal only):

```
?-phrase(sentence,[the,man,eats,the,apple]).
yes
```

Phrase Predicate

Definition of `phrase` is easy

```
phrase(Predicate,Argument):-
        Goal=..[Predicate,Argument,[]],
        call(Goal).
```

`=..` (read "equiv") – built-in predicate

=..

```
?- p(a,b,c)=..X.
X = [p, a, b, c]

?- X=..p(a,b,c).
ERROR: Type error: `list' expected, found `p(a, b, c)'

?- X=..[p,a,b,c].
X=p(a,b,c).

?- X=..[].
ERROR: Domain error: `not_empty_list' expected, found `[]'

?- X=..[1,a].
ERROR: Type error: `atom' expected, found `1'
```

Is Not It Enough?

No, we want more.

Distinguish singular and plural sentences.

Ungrammatical:

The boys eats the apple
The boy eat the apple

Straightforward Way

Add more grammar rules:

```
sentence    --> singular_sentence.
sentence    --> plural_sentence.

noun_phrase --> singular_noun_phrase.
noun_phrase --> plural_noun_phrase.

singular_sentence --> singular_noun_phrase,
                      singular_verb_phrase.

singular_noun_phrase --> singular_determiner,
                         singular_noun
```

```
singular_verb_phrase --> singular_verb, noun_phrase
singular_verb_phrase --> singular_verb

singular_determiner  --> [the]

singular_noun        --> [man]
singular_noun        --> [apple]

singular_verb        --> [eats]
singular_verb        --> [sings]
```

And similar for plural phrases.

Disadvantages

Not elegant

Obscures the fact that singular and plural sentences have a lot of structure in common.

Better solution:

Associate an extra argument to phrase types

According to whether it is singular or plural

```
sentence(singular)
sentence(plural)
```

Grammar Rules with Extra Arguments

```
sentence        --> sentence(X).
sentence(X)     --> noun_phrase(X), verb_phrase(X).

noun_phrase(X) --> determiner(X), noun(X).

verb_phrase(X) --> verb(X), noun_phrase(Y).
verb_phrase(X) --> verb(X).

determiner(_)  --> [the].

noun(singular) --> [man].
noun(singular) --> [apple].
```

```
noun(plural) --> [men].
noun(plural) --> [apples].

verb(singular) --> [eats].
verb(singular) --> [sings].

verb(plural) --> [eat].
verb(plural) --> [sing].
```

Parse Tree

`The man eats the apple`

Generates

```
sentence(
   noun_phrase(
        determiner(the),
        noun(man)),
   verb_phrase(
        verb(eats),
        noun_phrase(
           determiner(the),
           noun(apple)),
              )
         )
```

Building Parse Trees

We might want grammar rules to make a parse tree as well.

Rules need one more argument.

The argument should say how the parse tree for the whole phrase can be constructed from the parse trees of its sub-phrases.

Example:

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

Translation

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

translates to

```
sentence(X,sentence(NP,VP),S0,S) :-
noun_phrase(X,NP,S0,S1), verb_phrase(X,VP,S1,S).
```

Grammar Rules for Parse Trees

Number agreement arguments are left out for simplicity.

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

verb_phrase(verb_phrase(V)) -->
      verb(V).
verb_phrase(verb_phrase(VP,NP)) -->
      verb(VP),
      noun_phrase(NP).

noun_phrase(noun_phrase(DT,N)) -->
      determiner(DT),
      noun(N).
```

```
determiner(determiner(the)) --> [the].

noun(noun(man)) --> [man].
noun(noun(apple)) --> [apple].

verb(verb(eats)) --> [eats].
verb(verb(sings)) --> [sings].
```

Adding Extra Rules

So far everything in the grammar rules were used in processing the input sequence.

Every goal in the translated Prolog clauses has been involved with consuming some amount of input.

Sometimes we may want to specify Prolog clauses that are not of this type.

Grammar rule formalism allows this.

Overhead in Introducing New Word

To add a new word `banana`, add at least one extra rule:

```
noun(singular, noun(banana)) --> [banana].
```

Translated into Prolog:

```
noun(singular, noun(banana), [banana|S],S).
```

Too much information to specify for one noun.

Mixing Grammar with Prolog

Can not we put common information about all words in one place, and info about particular words in somewhere else?

Yes

```
noun(S, noun(N)) --> [N],{is_noun(N,S)}.
is_noun(banana,singular).
is_noun(banana,plural).
is_noun(man,singular).
```

Mixing Grammar with Prolog

```
noun(S, noun(N)) --> [N],{is_noun(N,S)}.
```

`{is_noun(N,S)}` is a test (condition).

`N` must be in the `is_noun` collection with some plurality `S`.

Curly brackets indicate that it expresses a relation that has nothing to do with the input sequence.

Translation does not affect expressions in the curly brackets:

```
noun(S, noun(N),[N|Seq],Seq):-is_noun(N,S).
```

Mixing Grammar with Prolog

Another inconvenience:

```
is_noun(banana,singular).
is_noun(banana,plural).
```

Two clauses for each noun.

Can be avoided in most of the cases
by adding `s` for plural at the and of singular.

Mixing Grammar with Prolog

Amended rule:

```
noun(plural, noun(RootN)) -->
    [N],
    {(name(N,Plname),
    append(Singname,"s",Plname),
    is_noun(RootN,singular))} .
```

Further Extension

So far the rules defined things
in terms how the input sequence is *consumed*.

We might like to define things
that *insert* items into the input sequence.

Example: analyze

“Eat your supper”

as if there were an extra word “you” inserted:

“You eat your supper”

Rule for the Extension

```
sentence --> imperative, noun_phrase, verb_phrase.
imperative, [you] --> [].
imperative --> [].
```

The first rule of `imperative` translate to:

```
imperative(L,[you|L]).
```

Meaning of the Extension

If

the left hand side of a grammar rule consists of

a part of the input sequence

separated from a list of words by comma

then

in the parsing, the words are inserted into the input sequence

after the goals on the right-hand side have had their chances

to consume words from it.