MAU11602

Lecture 3

2026-01-28

Natural languages

The way we describe formal languages, called generative grammar, originated with natural languages, e.g. Sanskrit, English, Irish, Japanese, Toki Pona.

Here's a vastly oversimplified grammar for English:

```
clause ::= np vp | clause pp
np     ::= NOUN | DET NOUN | np pp
pp     ::= PREP np
vp     ::= VERB np
```

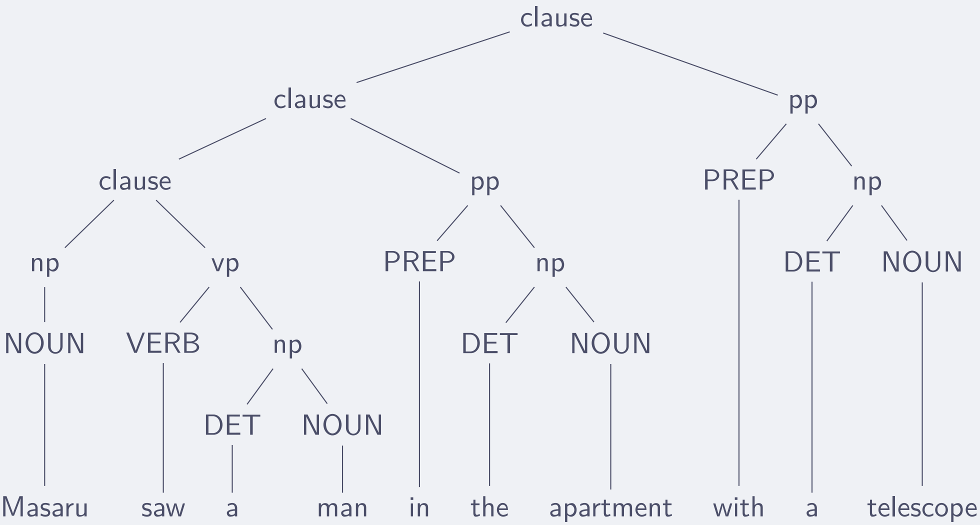
This requires an initial lexing phrase to identify `NOUN`s, `DET`s, `PREP`s, and `VERB`s.

This is not really possible in English. Is saw a noun or a verb?

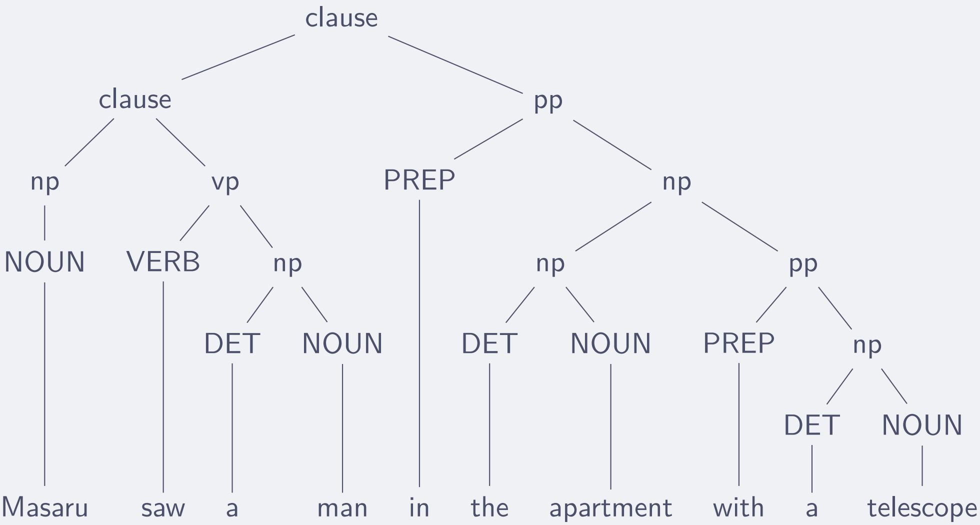
Even if we lex all the words correctly, parsing English is still highly ambiguous.

How do you interpret the headline Scientists discover emperor penguin colony in Antarctica using satellite images?

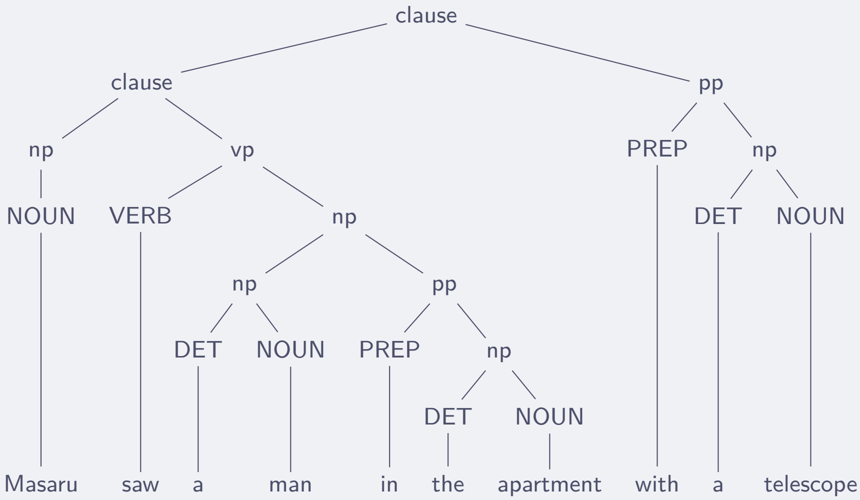
Example parse tree



Another possible tree



Yet another possible tree



A grammar for arithmetic with relations and boolean operators

```
  expr ::= bexpr0 | iexpr0
bexpr0 ::= "if" ws bexpr0 ws "then" ws bexpr0 ws "else" ws bexpr0
         | bexpr1
bexpr1 ::= bexpr2 ws "orelse" ws bexpr1 | bexpr2
bexpr2 ::= bexpr3 ws "andalso" ws bexpr2 | bexpr3
bexpr3 ::= "(" bexpr0 ")" | iexpr0 rel iexpr0 | "true" | "false"
   rel ::= "<" | "<=" | "=" | ">=" | ">"
iexpr0 ::= "if" ws bexpr0 ws "then" ws iexpr0 ws "else" ws iexpr0
         | iexpr0 ws addop ws iexpr0 | iexpr1
iexpr1 ::= iexpr1 ws mulop ws iexpr2 | iexpr2
iexpr2 ::= "(" iexpr0 ")" | int
```

I've added boolean operators and also conditionals. The rules for these are standard, but complicated. `orelse` has lower precedence than `andalso` but higher precedence than conditionals.

Unlike integer operators, which are left associative, the boolean operators are right associative. That doesn't matter yet, but will later.

Reduction semantics

The grammar does not specify the meaning of anything. For that we need to detail how to reduce expressions.

`false` and `true` are values.

`if false then` e_1 `else` e_2 reduces to e_2. `if true then` e_1 `else` e_2 reduces to e_1.

The relations <, <=, =, >, and >= reduce as you would expect them to, e.g. v_1 >= v_2 reduces to `false` if $v_1 < v_2$ and to `true` if $v_1 \geq v_2$. I've written v's here rather than e's because we will only reduce this expression after both subexpressions have been fully evaluated, i.e. reduced to values.

I never explicitly said what the reduction rules for +, −, and ∗ are, but they're defined similarly.

We also need what are called *congruence rules* to make this work, i.e. if e_1 reduces to e_2 then e_1 + e_3 reduces to e_2 + e_3 and v + e_1 reduces to v + e_2.

In other words, to evaluate a + expression we first evaluate the first subexpression to a value, then evaluate the second subexpression to a value, and then add the values.

Similar remarks apply to the other arithmetic operators and the relations, but not to conditionals!

Boolean operators

Why didn't I do the same for conditionals? Because if I did then it would force us to evaluate both branches, even though only one will be used.

`false orelse` *e* reduces to *e* and `true orelse` *e* reduces to `true`.

`false andalso` *e* reduces to `false` and `true andalso` *e* reduces to *e*.

So `orelse` is an inclusive or, not an exclusive or.

You might have expected `false orelse false`, `false andalso false`, `false andalso true`, and `true andalso false` to reduce to `false`, and `false orelse true`, `true orelse false`, `true orelse true`, and `true andalso true` to reduce to `true`.

Is this equivalent? Yes, and no. It wouldn't affect the value of any expression, but it would force us to fully evaluate the second subexpression in an `orelse` or `andalso` expression even when its value won't be used.

If you're one of those people who likes to divide integers then it will matter. The expression we don't evaluate might involve division by zero.

We often say things like if $x = 0$ or $y/x > 0$. We mean for the second branch of the if not to be evaluated when $x = 0$. In fact that's probably why we put the first branch in.