MAU22C00 Lecture 2

John Stalker

Trinity College Dublin

Example language: positive integers

Consider the language of positive integers, written in the usual decimal notation, without separating commas.

That is, “2023” is an element of the language but “2,023” and “002023” are not.

We can write a phrase structure grammar for this language:

```
%start pos_integer
%%
pos_integer : pos_digit | pos_integer digit ;
digit : 0 | pos_digit ;
pos_digit : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;
```

Positive integers, continued

Other grammars are possible, e.g.

```
%start pos_integer
%%
pos_integer : pos_digit | pos_digit digits ;
digits : digit | digit digits ;
digit : 0 | pos_digit ;
pos_digit : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;
```

The rule

```
pos_integer : pos_digit | pos_integer digit ;
```

from the first grammar is left recursive while the following rule from this grammar is right recursive:

```
digits : digit | digit digits ;
```

Ambiguity

The grammars we've seen so far are unambiguous. For any element of the language there's only one way to generate it, except for performing the same expansions in a different order.

Other grammars are ambiguous, like

```
%start statement
%%
statement : expression terminator ;
expression : animal
           | expression 'and' expression ;
animal : 'lions' | 'tigers' | 'bears' | 'quokkas' ;
terminator : 'oh' 'my' '!' ;
```

The things in quotes are tokens.

Example language: The Wizard of Oz

The statement "lions and tigers and bears oh my !" is an element of this language.

There are two different ways to generate it.

Both start by expanding `statement` to `expression terminator`, expanding the `terminator` in the only way allowed, and expanding the `expression` to `expression and expression`.

At that point we can expand the first `expression` to `animal` and then to `lions` and we can expand the second `expression` to `expression 'and' expression` again.

We then expand one `expression` to `animal` and `tigers` and the other to `animal` and then to `bears`.

The Wizard of Oz, continued

The other way to generate "lions and tigers and bears oh my !" expands the second `expression` to `bears` and the first one, via a few steps, to `lions and tigers`.

These are genuinely different parsings. In the first one "tigers and bears" occurs as a phrase by "lions and tigers" is not a phrase. In the second it's the opposite.

One way to exhibit which lists are phrases is with parentheses, e.g. ((lions and (tigers and bears)) (oh my !)) versus (((lions and tigers) and bears) (oh my !)).

Another way to describe parsings is with trees.

A parse tree

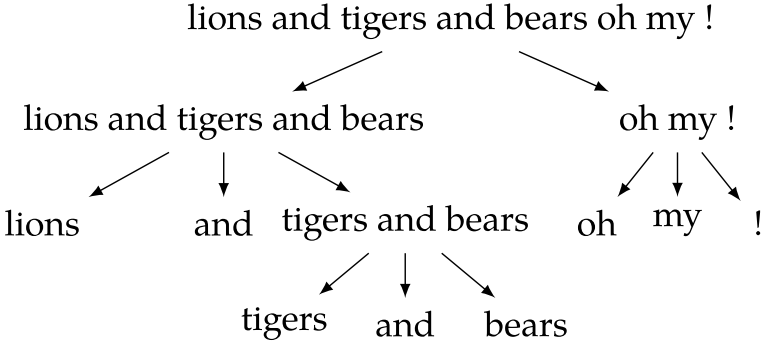

Figure 1: Parsing, first option

An alternate parse tree

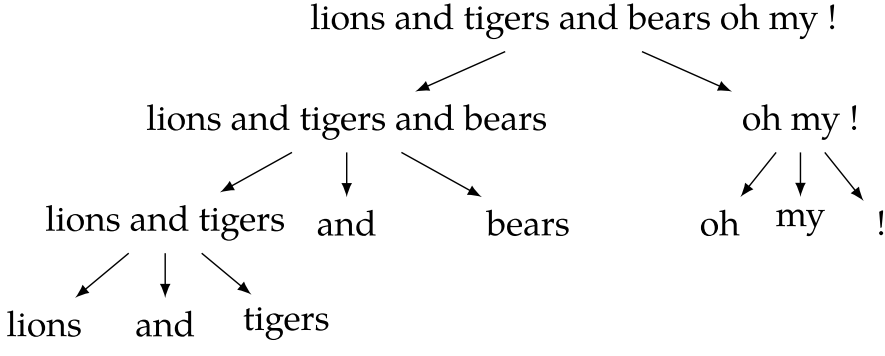

Figure 2: Parsing, first option

Ambiguity, continued

Some language are *inherently ambiguous*. It is impossible to construct an unambiguous grammar for them.

Sometimes an ambiguity is harmless. There's no difference in interpretation in the two parsings above.

Disambiguation, where possible, can seem unnatural. Which of these is clearest?

```
expression : animal | expression 'and' animal ;
```

```
expression : animal | animal 'and' expression ;
```

```
expression : animal | expression 'and' expression ;
```

I tend to write unambiguous grammars by habit.

Example language: palindromes

Palindromes are words or sentences which are the unchanged by reversing the letters.

Usually we ignore capitalisation, spaces and punctuation, so “Anna”, “Bob” and “Otto” are palindromes, as are “Madam, I’m Adam” and “NIΨON ANOMHMATA MH MONAN OΨIN”.

A grammar for palindromes is

```
%start S1
%%
S1 : /* empty */ | S2 ;
S2 : a S2 a | b S2 b | ... | z S2 z
   | a a | b b | ... | z z
   | a | b | ... | z ;
```

Example languages: zeroeth order logic

Our next topic is zeroeth order logic, a.k.a. the propositional calculus. There is a language which, with variations, is traditionally used for statements in ZOL.

The tokens are single characters, at least in unicode. We have

- variables p, q, r, etc.
- the unary "not" operator $\neg$.
- some subset of the binary operators $\wedge$, $\vee$, $\supset$, $\overline{\wedge}$, $\veebar$, $\equiv$, $\not\equiv$, and $\subset$. The first three represent "and", "or" and "implies".
- parentheses, square brackets and braces for grouping.

A grammar for this language is on the next slide.

Zeroeth order logic, continued

```
%start statement
%%
statement  : expression ;
expression : variable
           | ( expression binop expression )
           | [ expression binop expression ]
           | { expression binop expression }
           | ( ¬ expression )
           | [ ¬ expression ]
           | { ¬ expression } ;
variable   : p | q | r | s | u ;
binop      : ∧ | ∨ | ⊃ | ⊼ | ⊻ | ≡ | ≢ | ⊂ ;
```

Syntax versus semantics

Formal languages have only syntax, not semantics.

In other words, we can ask whether a list of tokens belongs to the language and, if it has a grammar, how it can be parsed, but not what it means.

Languages often have one or more intended interpretations, and additional, unintended interpretations.

Our language for linear equations has an interpretation where the variables are understood to be rational numbers, one where they're understood to be real, and one where they're understood to be complex. Being able to use the same language with these three interpretations is useful.

The language of positive integers has an alternate interpretation as positive hexadecimal integers which don't contain the digits `A`, `B`, `C`, `D`, `E` or `F`.

The Wizard of Oz language and palindrome language don't have any obvious interpretation.

Syntax versus semantics, continued

The language of balanced parentheses has a (surprising?) interpretation in terms of stack operations. (corresponds to *push* and) corresponds to *pop*. Elements of the language are permitted sequences of operations which leave an empty stack if you started with an empty stack.

A stack is a data structure which allows us to push items onto it, pop items onto it, and query whether it's empty.

The stack always has a finite number of elements, but there is no upper bound for this number.

The language for zeroeth order logic has an intended interpretation, where the Boolean operators mean what you think they mean.

Strictly, this is a class of interpretations, where the different ways to assign values to the variables are each a different interpretation.

There are other interpretations though.

More interpretations of the ZOL language

We could consider the variables p, q, ... as taking the values 0 or 1, interpret $(p \wedge q)$ as the maximum of p and q, $(p \vee q)$ as their minimum, and $(\neg p)$ as $1 - p$.

In some sense this is the same interpretation, if we identify 0 with true statements and 1 with false statements.

This is a slightly repackaged version of Boole's insight that you can arithmeticise logic.

We could also have interpreted $(p \wedge q)$ as the minimum of p and q, $(p \vee q)$ as their maximum, and identified 0 with false statements and 1 with true statements.

Different interpretations can make true statements false and vice versa!

There's also an interpretation in terms of voltage levels in digital circuits. That's why computers exist.