Language Processors F29LP2, Lecture 3

Jamie Gabbay

February 2, 2014

Ambiguous grammars

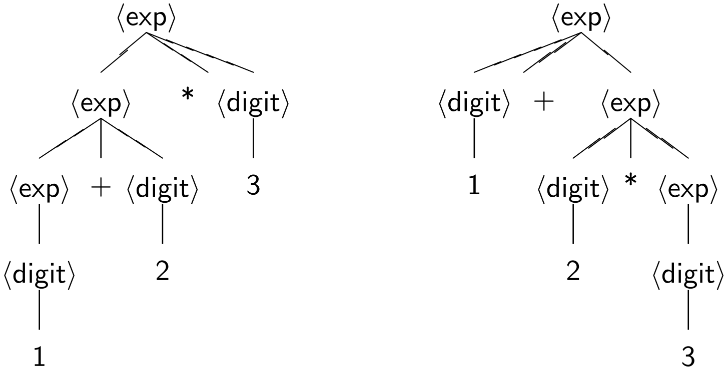
Call a grammar ambiguous when there is one sentence in the language with two parse trees.

For example consider this grammar:

$$\begin{array}{rcl} \langle\text{exp}\rangle & ::= & \langle\text{digit}\rangle \mid \langle\text{exp}\rangle{*}\langle\text{digit}\rangle \mid \langle\text{digit}\rangle{+}\langle\text{exp}\rangle \\ \langle\text{digit}\rangle & ::= & 1 \mid 2 \mid 3 \end{array}$$

Can it produce $1 + 2 * 3$ in two different ways?

Different parse trees



Parse trees and meaning

Different parse trees tend to give the same sentence different meanings.

For example $1 + 2 \quad * 3$ means 9, but $1 + \quad 2 * 3$ means 7.

So we come to the end of a journey; from raw program source code

```
72 7B 62 6C 75 65 7D 7B 23 31 7D 7D 0A 0A 5C 6E 65 77 63
6F 6D 6D 61 6E 64 7B 5C 47 72 65 65 6E 7D
```

to parsed code.

My question was: what is program source code?

My answer is: parsed syntax.

Unambiguous grammars

The problem "Is this grammar ambiguous" is undecidable. Who knows what an undecidable problem is?

Unambiguous grammars

That means that no computer program exists which inputs a grammar and:

- outputs ‘yes’ if it is ambiguous and
- outputs ‘no’ otherwise.

Unambiguous grammars

It may be best to write unambiguous grammars; grammars that for every sentence produce at most one parse tree.

Humans seem to produce grammars which are easy to disambiguate (rewrite in unambiguous form).

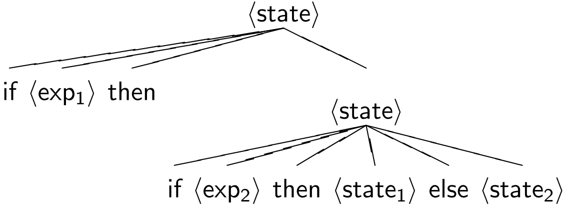
Consider the production rules

$$
\begin{aligned}
\langle \text{state} \rangle \quad ::= \quad & \text{if } \langle \text{exp} \rangle \text{ then } \langle \text{state} \rangle \mid \\
& \text{if } \langle \text{exp} \rangle \text{ then } \langle \text{state} \rangle \\
& \text{else} \langle \text{state} \rangle \mid \\
& \ldots
\end{aligned}
$$

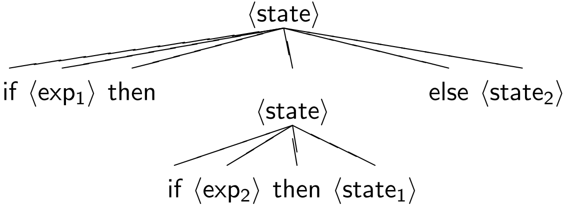
Now consider the sentential form (= string possibly mentioning non-terminals)

$$
\begin{aligned}
\text{if } \langle \text{exp}_1 \rangle \text{ then } \text{if } \langle \text{exp}_2 \rangle \quad & \text{then } \langle \text{state}_1 \rangle \\
& \text{else } \langle \text{state}_2 \rangle
\end{aligned}
$$

Derivation 1



Production rules



Derivations 1 and 2 mean different things

$$\text{if}\langle\text{exp}_1\rangle\text{then if}\langle\text{exp}_2\rangle \quad \begin{array}{l}\text{then}\langle\text{state}_1\rangle \\ \text{else}\langle\text{state}_2\rangle\end{array}$$

Is $\langle\text{state}_2\rangle$ executed when $\langle\text{exp}_1\rangle$ is true and $\langle\text{exp}_2\rangle$ is false
. . . or. . .
is $\langle\text{state}_2\rangle$ executed when $\langle\text{exp}_1\rangle$ is false?

Eliminate ambiguity

We usually match each else with the closest previous unmatched then.

So we prefer derivation 1.

How do we formalise this in a grammar?

Eliminate ambiguity

$$
\begin{aligned}
\langle \text{state} \rangle \quad & ::= \quad \langle \text{mstate} \rangle \mid \langle \text{umstate} \rangle \\
\langle \text{mstate} \rangle \quad & ::= \quad \text{if} \langle \text{exp} \rangle \text{then} \langle \text{mstate} \rangle \\
& \qquad\;\; \text{else} \langle \text{mstate} \rangle \\
\\
\langle \text{umstate} \rangle \quad & ::= \quad \text{if} \langle \text{exp} \rangle \text{then} \langle \text{state} \rangle \mid \\
& \qquad\;\; \text{if} \langle \text{exp} \rangle \text{then} \langle \text{mstate} \rangle \\
& \qquad\;\; \text{else} \langle \text{umstate} \rangle
\end{aligned}
$$

Recursive grammars

Call a grammar recursive when its productions are of the form:

$$A ::= \ldots A \ldots.$$

Most useful grammars are recursive.

Left-recursive grammars contain productions of the form $A ::= \ldots \mid A\alpha \mid \ldots.$

Right-recursive grammars contain productions of the form $A ::= \ldots \mid \alpha A \mid \ldots.$

(A grammar can be simultaneously left- and right-recursive.)

Example left-recursive grammar

$$\begin{aligned}
\langle \text{exp} \rangle &::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle \\
\langle \text{term} \rangle &::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &::= (\langle \text{exp} \rangle) \mid \text{id}
\end{aligned}$$

Problems with left-recursive grammars

Consider the grammar $S ::= aS \mid aT \qquad T ::= bT \mid b.$

We can check whether *aab* is in the language defined by this grammar by trying to produce two *a*s and one *b*. We can think of this as 'eating' the string. Each production rule 'eats' an *a* or a *b*.

$$aab \overset{S::=aS}{\Longrightarrow} ab \overset{S::=aT}{\Longrightarrow} b \overset{T::=b}{\Longrightarrow} \epsilon$$

Constructing a parser for a left-recursive grammar in this style, can be problematic. For example, for a top-down parser which expands left-to-right (leftmost derivation), left-recursion leads to non-termination. Suppose A is the start symbol. Then a left-recursive production rule $A ::= A\alpha$ leads to this recursive loop:

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow \ldots$$

Problems with left-recursive grammars

Likewise,

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$

may lead to

$$\begin{aligned} \langle \text{exp} \rangle &\Rightarrow \langle \text{exp} \rangle + \langle \text{term} \rangle \\ &\Rightarrow \langle \text{exp} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle \ldots \end{aligned}$$

We may re-phrase a grammar to eliminate left-recursion, if we want to mechanise parsing.

Otherwise, the moment we hit 'generate all sentences' the grammar will just loop.

How to eliminate left-recursion?

Question: Given a grammar G, how to define an equivalent (= they generate the same language) grammar G' which is not left-recursive?

Answer: Transform G in steps. Let the productions for A be

$$A ::= A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

where $\beta_1, \ldots, \beta_m$ do not begin with A.

Replace with productions

$$\begin{array}{l} A ::= \beta_1 \mid \beta_1 A' \mid \ldots \mid \beta_m \mid \beta_m A' \\ A' ::= \alpha_1 \mid \alpha_1 A' \mid \ldots \mid \alpha_n \mid \alpha_n A' \end{array}$$

where A' is a fresh nonterminal.

Run the algorithm

Before:

$$
\begin{aligned}
\langle \text{exp} \rangle &::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle \\
\langle \text{term} \rangle &::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &::= (\langle \text{exp} \rangle) \mid \text{id}
\end{aligned}
$$

Run the algorithm

After:

$$
\begin{aligned}
\langle \text{exp} \rangle &::= \langle \text{term} \rangle \langle \text{exp}' \rangle \\
\langle \text{exp}' \rangle &::= +\langle \text{term} \rangle \langle \text{exp}' \rangle \mid \varepsilon \\
\langle \text{term} \rangle &::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle &::= *\langle \text{factor} \rangle \langle \text{term}' \rangle \mid \varepsilon \\
\langle \text{factor} \rangle &::= (\langle \text{exp} \rangle) \mid \text{id}
\end{aligned}
$$

Top-down parsing (left-most derivation)

Construct a left-most derivation of id + id using the 'before' grammar:

$$
\begin{aligned}
\langle \mathrm{exp} \rangle &\Rightarrow \langle \mathrm{exp} \rangle + \langle \mathrm{term} \rangle \\
&\Rightarrow \langle \mathrm{exp} \rangle + \langle \mathrm{term} \rangle + \langle \mathrm{term} \rangle \\
&\Rightarrow \ldots
\end{aligned}
$$

Top-down parsing (left-most derivation)

Construct a left-most derivation of id + id using the 'after' grammar:

$$
\begin{array}{lcl}
\langle \text{exp} \rangle & \Rightarrow & \langle \text{term} \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \langle \text{factor} \rangle \langle \text{term}' \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} \langle \text{term}' \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} + \langle \text{term} \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} + \langle \text{factor} \rangle \langle \text{term}' \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} + \text{id} \langle \text{term}' \rangle \langle \text{exp}' \rangle \\
& \Rightarrow & \text{id} + \text{id}
\end{array}
$$

Nondeterminism

Do all grammars run along the same path always doing the same thing? Not necessarily.

Call a grammar nondeterministic when it contains a pair of productions

$$A ::= ab\alpha \mid ab\beta.$$

So two rewrites from A are possible: to $ab\alpha$, or to $ab\beta$.

This may give rise to backtracking, and is therefore bad news for algorithms.

Who knows what backtracking is?

Eliminate nondeterminism

Given a production

$$A ::= aB \mid aC$$

and a input string of the form $a\alpha$ we can not tell which branch to explore; $aB\alpha$ or $aC\alpha$.

Transform the production thus:

$$A ::= aA' \qquad A' ::= B \mid C.$$

This is known as left-factoring.

It eliminates the nondeterministic choice.

Example of left-factoring

Define an if-statement by:

$$\begin{array}{rcl}\langle\text{if-state}\rangle & ::= & \text{if}\langle\text{exp}\rangle\text{then}\langle\text{state}\rangle \mid \\ & & \text{if}\langle\text{exp}\rangle\text{then}\langle\text{state}\rangle \\ & & \text{else}\langle\text{state}\rangle\end{array}$$

Suppose I give you some target string. You want to generate it using left-most derivation. So you start with $\langle\text{exp}\rangle$ and try to produce it. You come to your first if in the string you want to produce.

Which production to apply — with else, or without else?

Example of left-factoring

You might choose the else production when you shouldn't, because the target string does not mention else (but you're examining it from left to right, remember).

So either you look ahead to anticipate problems, or you don't look ahead and prepare to backtrack.

This may be complex and inefficient.

Example of left-factoring

Or you use left-factoring. Change the grammar to this:

$$
\begin{array}{lcll}
\langle \text{if-state} \rangle & ::= & \text{if} \langle \text{exp} \rangle \text{then} & \langle \text{state} \rangle \\
 & & & \langle \text{if-rhs} \rangle \\
\langle \text{if-rhs} \rangle & ::= & \text{else} \langle \text{state} \rangle \mid \varepsilon &
\end{array}
$$

In summary:

Grammars can be

- ambiguous (so you don't know what something means),
- recursive (so your parser may loop), or
- non-deterministic (so your parser backtracks).

Grammars are, in fact, a programming language; production rules are commands that are nondeterministically executed.

Grammar transformations transform grammars to make them easier to implement and/or to make them execute more efficiently.