

Context-free grammars

Informatics 2A: Lecture 9

John Longley

School of Informatics
University of Edinburgh
jrl@inf.ed.ac.uk

7 October 2016

Recap of lecture 7

- Languages that require an ability to ‘count’ are not regular.
- Examples of this are $\{a^n b^n \mid n \geq 0\}$ and the language of well-matched sequences of brackets.
- The **pumping lemma** captures a pattern of regularity necessarily present in a regular language.
- When applied in its **contrapositive form** the pumping lemma provides a powerful tool for proving that a given language is not regular.

Beyond regular languages

Regular languages have significant limitations. (E.g. they can't cope with nesting of brackets).

So we'd like some more powerful means of defining languages.

Today we'll explore a new approach — via **generative grammars** (Chomsky 1952). A language is defined by giving a set of rules capable of 'generating' all the sentences of the language.

The particular kind of generative grammars we'll consider are called **context-free grammars**.

Context-free grammars: an example

Here is an example **context-free grammar**.

$$\text{Exp} \rightarrow \text{Var} \mid \text{Num} \mid (\text{Exp})$$
$$\text{Exp} \rightarrow \text{Exp} + \text{Exp}$$
$$\text{Exp} \rightarrow \text{Exp} * \text{Exp}$$
$$\text{Var} \rightarrow x \mid y \mid z$$
$$\text{Num} \rightarrow 0 \mid \dots \mid 9$$

It generates simple **arithmetic expressions** such as

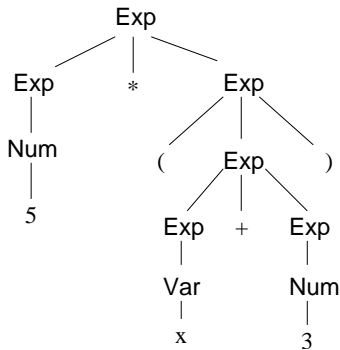
$6 + 7$ $5 * (x + 3)$ $x * ((z * 2) + y)$ 8 z

The symbols $+$, $*$, $($, $)$, x , y , z , 0 , \dots , 9 are called **terminals**: these form the ultimate constituents of the phrases we generate.

The symbols Exp , Var , Num are called **non-terminals**: they name various kinds of 'sub-phrases'. We designate Exp the **start symbol**.

Syntax trees

We grow **syntax trees** by repeatedly expanding non-terminal symbols using these rules. E.g.:



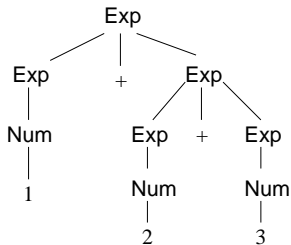
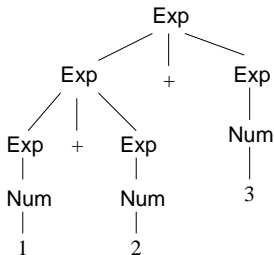
This generates $5 * (x + 3)$.

The language defined by a grammar

By choosing different rules to apply, we can generate infinitely many strings from this grammar.

The **language** generated by the grammar is, by definition, the set of all **strings of terminals** that can be derived from the **start symbol** via such a syntax tree.

Note that strings such as $1+2+3$ may be generated by more than one tree (**structural ambiguity**):



Challenge question

How many possible syntax trees are there for the string below?

$$1 + 2 + 3 + 4$$

Derivations

As a more 'machine-oriented' alternative to syntax trees, we can think in terms of **derivations** involving (mixed) strings of terminals and non-terminals. E.g.

$$\begin{aligned} \text{Exp} &\Rightarrow \text{Exp} * \text{Exp} \\ &\Rightarrow \text{Num} * \text{Exp} \\ &\Rightarrow \text{Num} * (\text{Exp}) \\ &\Rightarrow \text{Num} * (\text{Exp} + \text{Exp}) \\ &\Rightarrow 5 * (\text{Exp} + \text{Exp}) \\ &\Rightarrow 5 * (\text{Exp} + \text{Num}) \\ &\Rightarrow 5 * (\text{Var} + \text{Exp}) \\ &\Rightarrow 5 * (x + \text{Exp}) \\ &\Rightarrow 5 * (x + 3) \end{aligned}$$

At each stage, we choose one **non-terminal** and expand it using a suitable rule. When there are only **terminals** left, we can stop!

Multiple derivations

Clearly, any **derivation** can be turned into a **syntax tree**.

However, even when there's only one syntax tree, there might be many derivations for it:

Exp \Rightarrow Exp + Exp
 \Rightarrow Num + Exp
 \Rightarrow 1 + Exp
 \Rightarrow 1 + Num
 \Rightarrow 1 + 2

Exp \Rightarrow Exp + Exp
 \Rightarrow Exp + Num
 \Rightarrow Exp + 2
 \Rightarrow Num + 2
 \Rightarrow 1 + 2

(... a **leftmost** derivation)

(... a **rightmost** derivation)

In the end, it's the **syntax tree** that matters — we don't normally care about the differences between various derivations for it.

However, derivations — especially leftmost and rightmost ones — will play a significant role when we consider **parsing algorithms**.

Second example: comma-separated lists

Consider lists of (zero or more) alphabetic characters, separated by commas:

ϵ a e,d q,w,e,r,t,y

These can be generated by the following grammar (note the rules with *empty* right hand side).

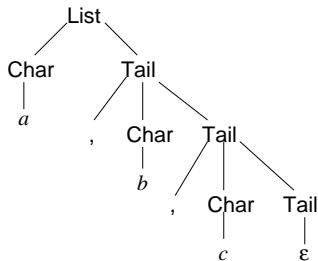
List $\rightarrow \epsilon \mid \text{Char Tail}$
Tail $\rightarrow \epsilon \mid , \text{Char Tail}$
Char $\rightarrow a \mid \dots \mid z$

Terminals: $a, \dots, z, ,$
Non-terminals: List, Tail, Char
Start symbol: List

Syntax trees for comma-separated lists

List $\rightarrow \epsilon \mid \text{Char Tail}$
Tail $\rightarrow \epsilon \mid , \text{Char Tail}$
Char $\rightarrow a \mid \dots \mid z$

Here is the syntax tree for the list a, b, c :



Notice how we indicate the application of an ' ϵ -rule'.

Other examples

- The language $\{a^n b^n \mid n \geq 0\}$ may be defined by the grammar:

$$S \rightarrow \epsilon \mid aSb$$

- The language of well-matched sequences of brackets () may be defined by

$$S \rightarrow \epsilon \mid SS \mid (S)$$

So both of these are examples of **context-free languages**.

Context-free grammars: formal definition

A **context-free grammar** (CFG) \mathcal{G} consists of

- a finite set N of **non-terminals**,
- a finite set Σ of **terminals**, disjoint from N ,
- a finite set P of **productions** of the form $X \rightarrow \alpha$, where $X \in N$, $\alpha \in (N \cup \Sigma)^*$,
- a choice of **start symbol** $S \in N$.

A **sentential form** is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

Formal definition: The set of **sentential forms** derivable from \mathcal{G} is the smallest set $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$ such that

- $S \in \mathcal{S}(\mathcal{G})$
- if $\alpha X \beta \in \mathcal{S}(\mathcal{G})$ and $X \rightarrow \gamma \in P$, then $\alpha \gamma \beta \in \mathcal{S}(\mathcal{G})$.

The **language** associated with grammar is the set of sentential forms that contain only terminals.

Formal definition: The **language** associated with \mathcal{G} is defined by $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A **sentential form** is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

Formal definition: The set of **sentential forms** derivable from \mathcal{G} is the smallest set $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$ such that

- $S \in \mathcal{S}(\mathcal{G})$
- if $\alpha X \beta \in \mathcal{S}(\mathcal{G})$ and $X \rightarrow \gamma \in P$, then $\alpha \gamma \beta \in \mathcal{S}(\mathcal{G})$.

The **language** associated with grammar is the set of sentential forms that contain only terminals.

Formal definition: The **language** associated with \mathcal{G} is defined by $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A language $L \subseteq \Sigma^*$ is defined to be **context-free** if there exists some CFG \mathcal{G} such that $L = \mathcal{L}(\mathcal{G})$.

Assorted remarks

- $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ is simply an **abbreviation** for a bunch of productions $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$.
- These grammars are called **context-free** because a rule $X \rightarrow \alpha$ says that an X can *always* be expanded to α , no matter where the X occurs.
This contrasts with **context-sensitive** rules, which might allow us to expand X only in certain contexts, e.g. $bXc \rightarrow bac$.
- Broad intuition: context-free languages allow **nesting of structures to arbitrary depth**. E.g. brackets, begin-end blocks, if-then-else statements, subordinate clauses in English, ...

Arithmetic expressions again

Our earlier grammar for **arithmetic expressions** was limited in that only single-character variables/numerals were allowed. One could address this problem in either of two ways:

- Add more grammar rules to allow generation of longer variables/numerals, e.g.

$$\begin{aligned}\text{Num} &\rightarrow 0 \mid \text{NonZeroDigit Digits} \\ \text{Digits} &\rightarrow \epsilon \mid \text{Digit Digits}\end{aligned}$$

- Give a separate description of the **lexical structure** of the language (e.g. using regular expressions), and treat the names of lexical classes (e.g. VAR, NUM) as **terminals** from the point of view of the CFG. So the CFG will generate strings such as

$$\text{NUM} * (\text{VAR} + \text{NUM})$$

The second option is generally preferable: lexing (using regular expressions) is computationally 'cheaper' than parsing for CFGs.

A programming language example

Building on our grammar for arithmetic expressions, we can give a CFG for a little programming language, e.g.:

stmt \rightarrow if-stmt | while-stmt | begin-stmt | assg-stmt
if-stmt \rightarrow **if** bool-expr **then** stmt **else** stmt
while-stmt \rightarrow **while** bool-expr **do** stmt
begin-stmt \rightarrow **begin** stmt-list **end**
stmt-list \rightarrow stmt | stmt ; stmt-list
assg-stmt \rightarrow VAR := arith-expr
bool-expr \rightarrow arith-expr compare-op arith-expr
compare-op \rightarrow < | > | <= | >= | == | != =

Grammars like this (often with ::= in place of \rightarrow) are standard in computer language reference manuals. This notation is often called **BNF** (Backus-Naur Form).

A natural language example

Consider the following lexical classes ('parts of speech') in English:

N	nouns	(<i>alien, cat, dog, house, malt, owl, rat, table</i>)
Name	proper names	(<i>Jack, Susan</i>)
TrV	transitive verbs	(<i>admired, ate, built, chased, killed</i>)
LocV	locative verbs	(<i>is, lives, lay</i>)
Prep	prepositions	(<i>in, on, by, under</i>)
Det	determiners	(<i>the, my, some</i>)

Now consider the following productions (start symbol S):

S	→	NP VP
NP	→	<i>this</i> Name Det N Det N RelCl
RelCl	→	<i>that</i> VP <i>that</i> NP TrV NP TrV
VP	→	<i>is</i> NP TrV NP LocV Prep NP

Natural language example in action

Even this modest bunch of rules can generate a rich multitude of English sentences, for example:

- *this is Jack*
- *some alien ate my owl*
- *Susan admired the rat that lay under my table*
- *this is the dog that chased the cat that killed the rat that ate the malt that lay in the house that Jack built*
- (?) *the malt the rat the cat the dog chased killed ate lay in the house*

(Hard to parse in practice — later we'll see 'why'.)

Nesting in natural language

Excerpt from Jane Austen, *Mansfield Park*.

Whatever effect Sir Thomas's little harangue might really produce on Mr. Crawford, it raised some awkward sensations in two of the others, two of his most attentive listeners — Miss Crawford and Fanny. One of whom, having never before understood that Thornton was so soon and so completely to be his home, was pondering with downcast eyes on what it would be *not* to see Edmund every day; and the other, startled from the agreeable fancies she had been previously indulging on the strength of her brother's description, no longer able, in the picture she had been forming of a future Thornton, to shut out the church, sink the clergyman, and see only the respectable, elegant, modernized and occasional residence of a man of independent fortune, was considering Sir Thomas, with decided ill-will, as the destroyer of all this, and suffering the more from . . .

More nesting in natural language

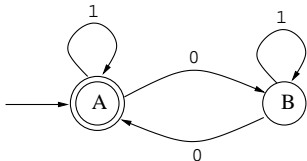
From a recent magazine article (2015):

I am not suggesting that the answer to the immense problems that the huge number of refugees from Syria and from oppression, political, religious or economic, across the globe are causing to countries faced by the initial arrival is an easy one . . .

(Canon John Richardson, St. Columba's-by-the-Castle church magazine, Harvest 2015)

Every regular language is context-free!

We can easily turn a DFA into a CFG, e.g.



$$A \rightarrow \epsilon \mid 1A \mid 0B$$

$$B \rightarrow 1B \mid 0A$$

Start symbol: A

- **Terminals** are **input symbols** for the DFA.
- **Non-terminals** are **states** of the DFA.
- **Start symbol** is **initial state**.
- For every **transition** $X \xrightarrow{a} Y$, we have a **production** $X \rightarrow aY$.
- For every **accepting state** X , we have a **production** $X \rightarrow \epsilon$.

A CFG is called **regular** if all rules are of the form $X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow \epsilon$. The languages definable by regular CFGs are precisely the **regular languages**.

End-of-lecture self-assessment question

Recall from Slide 11:

$$\text{List} \rightarrow \epsilon \mid \text{Char Tail}$$
$$\text{Tail} \rightarrow \epsilon \mid , \text{Char Tail}$$

Which of the following alternative context-free grammars for List is incorrect in the sense that it defines a different language for List?

1:
$$\begin{aligned} \text{List} &\rightarrow \epsilon \mid \text{Body Char} \\ \text{Body} &\rightarrow \epsilon \mid \text{Body Char} , \end{aligned}$$

2:
$$\begin{aligned} \text{List} &\rightarrow \epsilon \mid \text{NonEmpty} \\ \text{NonEmpty} &\rightarrow \text{Char} \mid \text{Char} , \text{NonEmpty} \end{aligned}$$

3:
$$\begin{aligned} \text{List} &\rightarrow \epsilon \mid \text{NonEmpty} \\ \text{NonEmpty} &\rightarrow \text{Char} \mid \text{NonEmpty} , \text{NonEmpty} \end{aligned}$$

4: They are all correct

Reading and prospectus

Relevant reading:

- Kozen chapters 19, 20
- Jurafsky & Martin, sections 12.1–12.3

Next time: What kinds of **machines** (analogous to DFAs or NFAs) correspond to context-free languages?