Context-free grammars

Informatics 2A: Lecture 8

Alex Simpson

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

4 October, 2012

Beyond regular languages

We've seen that regular languages have their limits (e.g. can't cope with nesting of brackets).

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

Here 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's a grammar for generating simple arithmetic expressions such as

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

Grammar rules:

$$\begin{aligned}
\text{Exp} &\rightarrow \text{Var} \mid \text{Num} \mid (\ \text{Exp}\ ) \\
\text{Exp} &\rightarrow \text{Exp} + \text{Exp} \\
\text{Exp} &\rightarrow \text{Exp} * \text{Exp} \\
\text{Var} &\rightarrow x \mid y \mid z \\
\text{Num} &\rightarrow 0 \mid \cdots \mid 9
\end{aligned}$$

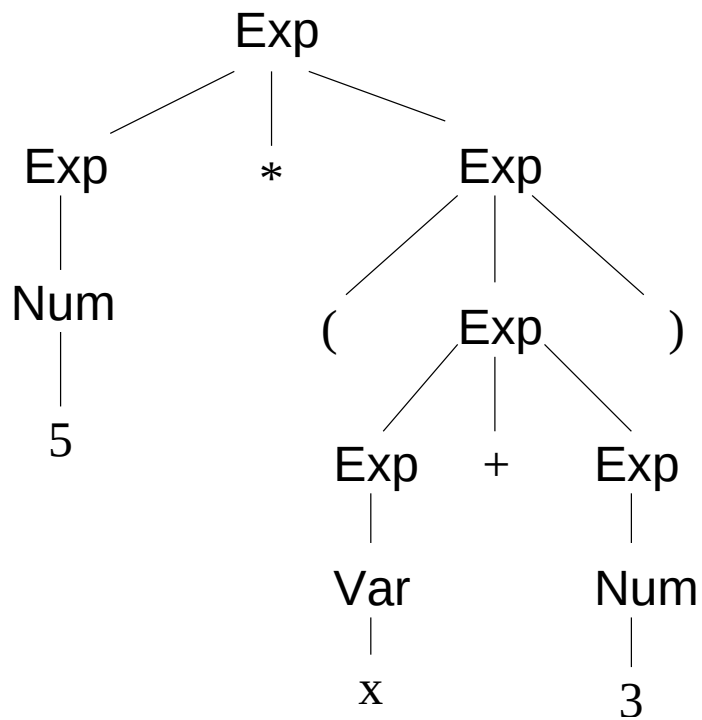
The symbols $+, *, (,), x, y, z, 0, \ldots, 9$ are called terminals: these form the ultimate constituents of the phrases we generate.

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

Syntax trees

$$\begin{array}{lcl} \text{Exp} & \to & \text{Var} \mid \text{Num} \mid (\text{Exp}) \\ \text{Exp} & \to & \text{Exp} + \text{Exp} \\ \text{Exp} & \to & \text{Exp} * \text{Exp} \end{array} \qquad \begin{array}{lcl} \text{Var} & \to & x \mid y \mid z \\ \text{Num} & \to & 0 \mid \cdots \mid 9 \end{array}$$

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



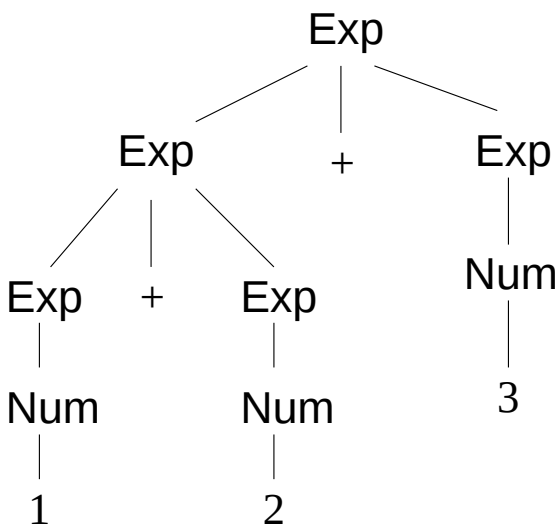
This generates $5 * (x + 3)$.

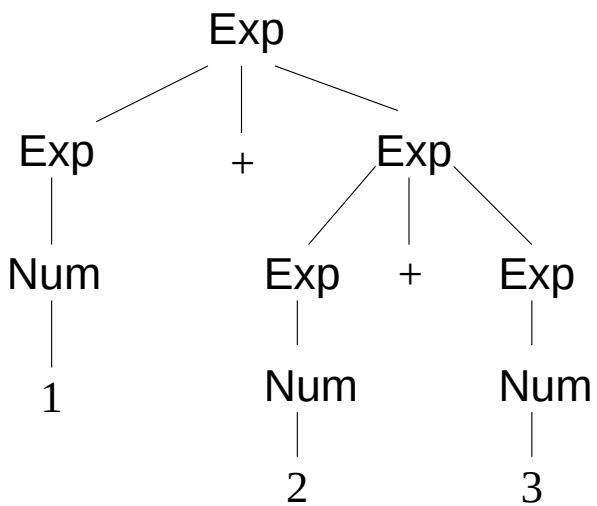
The language defined by a grammar

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):





Clicker question

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

$$1 + 2 + 3 + 4$$

A: 2

B: 3

C: 4

D: 5

E: > 5

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.

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

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:

$$\begin{aligned} \mathrm{Exp} &\Rightarrow \mathrm{Exp} + \mathrm{Exp} \\ &\Rightarrow \mathrm{Num} + \mathrm{Exp} \\ &\Rightarrow 1 + \mathrm{Exp} \\ &\Rightarrow 1 + \mathrm{Num} \\ &\Rightarrow 1 + 2 \end{aligned}$$

(. . . a leftmost derivation)

$$\begin{aligned} \mathrm{Exp} &\Rightarrow \mathrm{Exp} + \mathrm{Exp} \\ &\Rightarrow \mathrm{Exp} + \mathrm{Num} \\ &\Rightarrow \mathrm{Exp} + 2 \\ &\Rightarrow \mathrm{Num} + 2 \\ &\Rightarrow 1 + 2 \end{aligned}$$

(. . . 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:

$$\epsilon \qquad a \qquad e,d \qquad q,w,e,r,t,y$$

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

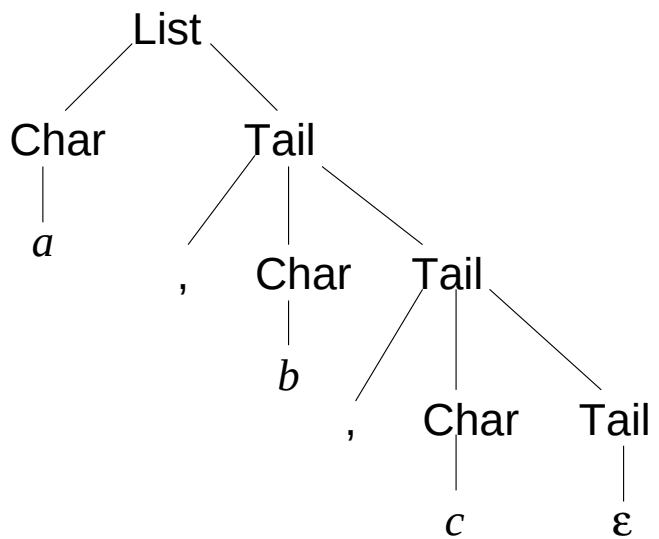
$$\begin{array}{rcl} \text{List} & \longrightarrow & \epsilon \mid \text{Char Tail} \\ \text{Tail} & \longrightarrow & \epsilon \mid \text{, Char Tail} \\ \text{Char} & \longrightarrow & a \mid \cdots \mid z \end{array}$$

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

Syntax trees for comma-separated lists

$$\begin{aligned} \text{List} &\to \epsilon \mid \text{Char Tail} \\ \text{Tail} &\to \epsilon \mid \text{, Char Tail} \\ \text{Char} &\to a \mid \cdots \mid z \end{aligned}$$

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



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

Clicker Question

$$\begin{aligned} \text{List} &\rightarrow \epsilon \mid \text{Char Tail} \\ \text{Tail} &\rightarrow \epsilon \mid \text{, Char Tail} \end{aligned}$$

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

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

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

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

D: They are all correct

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 $X \to \alpha$, where $X \in N$, $\alpha \in (N \cup \Sigma)^*$,
- a choice of start symbol $S \in N$.

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 $X \to \gamma \in P$, then $\alpha\gamma\beta \in \mathcal{S}(\mathcal{G})$.

The language associated with $\mathcal{G}$ is then defined as

$$\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$$

A language L is context-free if $L = \mathcal{L}(\mathcal{G})$ for some CFG $\mathcal{G}$.

Assorted remarks

- $X \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ is simply an abbreviation for a bunch of productions $X \to \alpha_1$, $X \to \alpha_2$, ..., $X \to \alpha_n$.
- These grammars are called context-free because a rule $X \to \alpha$ 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. $bXc \to b\alpha c$.
- 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 $\mid$ while-stmt $\mid$ begin-stmt $\mid$ 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 $\mid$ stmt ; stmt-list
assg-stmt	$\rightarrow$	VAR := arith-expr
bool-expr	$\rightarrow$	arith-expr compare-op arith-expr
compare-op	$\rightarrow$	$< \mid > \mid <= \mid >= \mid == \mid =! =$

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

Now consider the following productions (start symbol S):

$$
\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \textit{this} \mid \text{Name} \mid \text{Det N} \mid \text{Det N RelCl} \\
\text{RelCl} &\rightarrow \textit{that}\ \text{VP} \mid \text{NP TrV} \\
\text{VP} &\rightarrow \textit{is}\ \text{NP} \mid \text{TrV NP} \mid \text{LocV Prep NP}
\end{aligned}
$$

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 that Jack built*
  (Hard to parse in practice — later we'll see 'why'.)

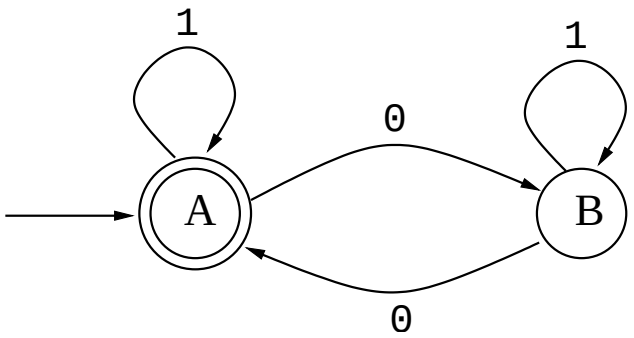
Nesting in natural language

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 . . .

And finally. . . regular implies context-free!

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



$$A \to \epsilon \mid 1\,A \mid 0\,B$$
$$B \to 1\,B \mid 0\,A$$

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 $X \to a\,Y$.
- For every accepting state X, we have a production $X \to \epsilon$.

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

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?