Chapter 7

Languages and grammars

A language is a set of possible sentences. Grammars describe languages by providing rules for constructing those possible sentences.

7.1 Grammars

A grammar is a set of rules. Each rule defines how to replace one pattern with another pattern.

\[
\text{left-hand side} \quad \text{right-hand side}
\]

original pattern $\rightarrow$ replacement pattern

Any given left-hand side can have one or more right-hand sides. Having multiple right-hand sides for a single left-hand side is how grammars achieve alternation.

Patterns contain two kinds of symbol. Terminal symbols are literals that appear directly in sentences of the language. Non-terminal symbols represent incomplete patterns that must be replaced before reaching a valid sentence. A pattern that contains both terminals and non-terminals is a sentential form. A pattern that contains only terminal symbols is a sentence (or sequence).

7.1.1 Backus-Naur Form

Backus-Naur Form (BNF) is one way to write grammar rules. Non-terminal symbols appear within curly-braces, e.g.: \(<\text{adjective}>\). Terminal symbols are written literally, or within quotation marks if ambiguity is possible, e.g: \(\text{simple}\) or “\(\text{simple}\)”. Alternative right-hand sides can be written as individual rules (repeating the same left-hand side) or as a single rule with alternative right-hand side patterns separated with ‘|’.

\[
<\text{adjective}> \quad \text{first} \\
<\text{adjective}> \quad \text{second} \\
or \\
<\text{adjective}> \quad \text{first} \mid \text{second}
\]

Common-sense can be used to simplify BNF rules.

\[
<\text{digit}> \quad 0 \mid 1 \mid 2 \mid \ldots \mid 7 \mid 8 \mid 9
\]
7.1.2 Producing sentences from the start rule

One of the rules (often the first) is designated as the start rule. To generate a valid sentence, an initial sentential form is created from the left-hand side of the start rule. A non-terminal symbol in the sentential form is then replaced with one of its right-hand sides. Replacement continues until only terminal symbols remain in the sentential form. This final sentential form is a valid sentence of the language described by the grammar.

The sequence of replacements of the non-terminals that convert the sentential form from the start rule into the final sentence is called the derivation of the sentence.

7.1.3 Repetition via recursion

Recursion occurs when a non-terminal symbol appears in the right-hand side of one of its own rules. Right (tail) recursion can be used to repeat a rule’s pattern indefinitely. A non-recursive alternative right hand side for the symbol prevents infinite recursion.

7.2 Regular grammars and finite state machines

Any finite state machine can be described by an equivalent grammar. Non-terminal symbols in the grammar are associated with FSM states. Any sequence of symbols that the FSM can produce from a given state is the same sequence of symbols that the grammar can produce from the corresponding non-terminal symbol. Each transition in the FSM creates a rule in the grammar.
All the rules in a grammar corresponding to a FSM therefore have one of the following forms.

\[
\begin{align*}
\langle P \rangle & \rightarrow \langle Q \rangle \quad (\text{an } \varepsilon\text{-transition } P \rightarrow Q) \\
\langle P \rangle & \rightarrow x \quad \langle Q \rangle \quad (\text{a labelled transition } P \xrightarrow{x} Q) \\
\langle P \rangle & \rightarrow y \quad (\text{a labelled transition } P \xrightarrow{y} \text{ final state}) \\
\langle P \rangle & \rightarrow \quad (\text{an } \varepsilon\text{-transition } P \rightarrow \text{ final state})
\end{align*}
\]

Any grammar in which all rules have one of these forms is a regular grammar. It has the same expressive power as a regular expression; i.e., it can generate a regular language that contains only regular sequences of symbols involving concatenation, alternation, and repetition.

### 7.2.1 Limitations of regular grammars

Regular grammars cannot describe languages that contain unbounded recursive constructs.

Expressions that contain parenthesised (sub-)expressions cannot be described by a regular grammar; the corresponding FSM would need unique states to represent each possible number of opening ‘(’s from which the corresponding closing ‘)’s can be matched.

Given a FSM (starting at S1) that recognises expressions surrounded by a single pair of parentheses, extending it to recognise expressions surrounded by two pairs of parentheses (starting at S2) can only be done by repeating and extending the original machine.

Matching unbounded pairs of parentheses using a fixed-sized machine is only possible if an unbounded memory is added to the machine.

### 7.3 Context-free grammars and push-down automata

Adding a stack to a finite-state machine produces a push-down machine (PDM) or push-down automaton (PDA). A PDM can count, by remembering past inputs independently of the particular state that happens to be current.

Transitions in a PDM can be conditional on the next input symbol (as in a FSM) and/or the topmost item (most-recently pushed) on the stack (including the absence of any item, when the stack is empty). Additionally, after choosing a transition, the machine can optionally push a new item onto the stack or pop the topmost item off the stack.
A bounded PDM that matches unbounded pairs of parentheses surrounding another sequence (e.g., ‘x’) is easily constructed.

Push-down machines can generate or recognise context-free languages, described by context-free grammars (CFGs). CFGs contain rules whose right-hand sides can be any sequence of terminal and/or non-terminal symbols, without restriction:

\[ \langle \text{non-terminal} \rangle \rightarrow \alpha \]

(where \( \alpha \) is any sequence of zero or more symbols). Expressions, including nested constructs such as parenthesised sub-expressions, are easily described by a CFG.

\[
\begin{align*}
\langle \text{expression} \rangle & \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle | \langle \text{term} \\ \langle \text{term} \rangle & \rightarrow ( \langle \text{expression} \rangle ) | \langle \text{number} \rangle
\end{align*}
\]

Almost all programming languages can be described by a context-free grammar.

### 7.4 Parsing

Parsing is the opposite of generation. Starting with a sentence, parsing determines whether or not that sentence can be generated by a given grammar. Parsing a sentence according to a grammar is a much more difficult problem than generating example sentences from that grammar.

In addition to recognising whether a sentence is valid, most parsers also provide a derivation for the sentence.

\[
\begin{align*}
\langle \text{identifier} \rangle & \rightarrow \langle \text{letter} \rangle \langle \text{identifier} \rangle | \langle \text{letter} \\
\langle \text{letter} \rangle & \rightarrow a | b | c | \ldots | x | y | z
\end{align*}
\]

production of “var”

derivation

\[
\begin{align*}
\langle \text{identifier} \rangle & \rightarrow \langle \text{letter} \rangle \langle \text{identifier} \rangle \\
\rightarrow & \langle \text{letter} \rangle \langle \text{identifier} \rangle \\
\rightarrow & \langle \text{letter} \rangle \langle \text{identifier} \rangle \\
\rightarrow & \langle \text{letter} \rangle \langle \text{identifier} \rangle \\
\rightarrow & \langle \text{letter} \rangle \langle \text{identifier} \rangle
\end{align*}
\]

Numeric subscripts on the non-terminal symbols identify the different instances of each symbol that appear within the sentential forms. At each step in the derivation, the left-hand side non-terminal symbol can be considered the ‘parent’ of the right-hand side symbols that it produces. (The right-hand side symbols are therefore the ‘children’ of their left-hand side non-terminal symbol.) The start symbol is the root of a parse tree, that shows the relationships between all of the symbols appearing in the derivation.
7.4.1 Ambiguity

Ambiguity arises when a given sentence can be generated in more than one way. Ambiguity is not significant when the grammar is \textit{generating} sentences to define a language. Conversely, ambiguity is highly significant when the grammar is being used to \textit{parse} a sentence whose derivation represents its precise meaning.

A grammar for expressions involving addition and multiplication can generate a sentence “$1+2*3$” in two different ways.

Operator precedence requires the multiplication operator to be applied before the addition operator, but only one of the two possible derivations represents the correct behaviour.

```
<expression>  *  <expression>
<expression>  +  <expression>
<expression>  =  <number>
<number>  +  <number>
<number>  *  <number>
<number>  -  <number>
```

Implicit parentheses

\[
(1+2)*3  \quad \text{and} \quad 1+(2*3)
\]

Splitting operators into distinct rules, nested according to their precedence, force derivations to describe the correct behaviour of arithmetic operators.

```
<addition>  \rightarrow  <addition>  +  <multiplication>  |  <multiplication>
<multiplication>  \rightarrow  <multiplication>  *  <number>  |  <number>
```

Using left recursion for repeated application of an operator gives it left-associative behaviour. In cases where right associativity is required (e.g., “$a=b=3+4$”) right-recursive rules produce correct derivations.

```
<assignment>  \rightarrow  <identifier>  =  <assignment>  |  <addition>
```

Syntactic structure is therefore related to the semantics (meaning) of a sentence.

7.5 Abstract syntax trees

Removing irrelevant syntactic details from the parse tree gives us an \textit{abstract syntax tree} (AST). For the sentence “$1+2*3$”, the only important information is the nature and identity of the two binary operators, and the kind of operands represented by the leaves of the tree.

Syntax trees are usually generated (instead of a parse tree) during parsing. For programming languages, if there are no ambiguities in the grammar, it is straightforward to use the AST to interpret (or to generate machine code for) the program it represents.