Languages, Grammars, and Graphs

Jim Hefferon

University of Vermont

hefferon.net

Languages

Strings

Recall that a symbol, or token, is something that a machine can read and write, and an alphabet is a nonempty and finite set of symbols.

A string or word over an alphabet is a finite sequence of elements from that alphabet. The string with no elements is the empty string, denoted ε, along with https://www.youtube.com/watch?v=juXwu0Nqc3I.

We write symbols in a distinct typeface, as in 1 or a, because the alternative of quoting them would be clunky.

Traditionally, we denote an alphabet with the Greek letter Σ. In this book we will name strings with lower case Greek letters (except that we use ϕ for something else) and denote the items in the string with the associated lower case roman letter, as in $\sigma = \langle s_0, \ldots s_{n-1} \rangle$ and $\tau = \langle t_0, \ldots t_{m-1} \rangle$. The length of the string σ, $|\sigma|$, is the number of symbols that it contains, n. In particular, the length of the empty string is $|\varepsilon| = 0$. We usually work with alphabets having single-character symbols and then we write strings by omitting the brackets and commas. That is, we write $\sigma = \mathtt{abc}$ instead of $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$.

Example Natural numbers are represented by strings of digits. The alphabet is $\Sigma = \{\mathtt{0}, \ldots \mathtt{9}\}$. The string $\sigma = \mathtt{1066}$ has length $|\sigma| = 4$.

The alphabet consisting of the bit characters is $\mathbb{B} = \{\mathtt{0}, \mathtt{1}\}$. Strings over this alphabet are bitstrings or bit strings.

String operations

Let $\sigma = \langle s_0 \ldots s_{n-1} \rangle$ and $\tau = \langle t_0, \ldots t_{m-1} \rangle$ be strings over an alphabet Σ. The concatenation $\sigma \frown \tau$ or $\sigma\tau$ appends the second string to the first, $\sigma \frown \tau = \langle s_0 \ldots s_{n-1}, t_0, \ldots t_{m-1} \rangle$. Where $\sigma = \tau_0 \frown \cdots \frown \tau_{k-1}$, we say that σ decomposes into the τ's, and that each τ_i is a substring of σ. The first substring, τ_0, is a prefix of σ. The last, τ_{k-1}, is a suffix.

A power or replication of a string is an iterated concatenation with itself, so that $\sigma^2 = \sigma \frown \sigma$ and $\sigma^3 = \sigma \frown \sigma \frown \sigma$, etc. We write $\sigma^1 = \sigma$ and $\sigma^0 = \varepsilon$. The reversal σ^{R} of a string takes the symbols in reverse order: $\sigma^{\mathsf{R}} = \langle s_{n-1}, \ldots s_0 \rangle$. The empty string's reversal is $\varepsilon^{\mathsf{R}} = \varepsilon$.

A palindrome is a string that equals its own reversal. Examples are $\alpha = \mathtt{abba}$, $\beta = \mathtt{cdc}$, and ε.

Where Σ is an alphabet, for $k \in \mathbb{N}$ the set of length k strings over that alphabet is Σ^k. The set of strings over Σ of any finite length is $\Sigma^* = \cup_{k \in \mathbb{N}} \Sigma^k$. The asterisk symbol is the Kleene star, read aloud as "star."

Example The set of bitstrings of length 3 is $\mathbb{B}^3 = \{ \mathtt{000}, \mathtt{001}, \ldots \mathtt{111} \}$. The set of all bitstrings is $\mathbb{B}^* = \{ \varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{00}, \ldots \}$.

Languages

DEFINITION A language $\mathcal{L}$ over an alphabet Σ is a set of strings drawn from that alphabet. That is, $\mathcal{L} \subseteq \Sigma^*$.

EXAMPLE Let $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. One language is $\mathcal{L}_0 = \{\sigma \in \Sigma^* \mid \sigma = \mathsf{a}^n\mathsf{b}\mathsf{a}^n \text{ for some } n \in \mathbb{N}\}$; some members are b, aba, aabaa, and aaabaaa. Another is the language of palindromes, $\mathcal{L}_1 = \{\sigma \in \Sigma^* \mid \sigma = \sigma^{\mathsf{R}}\}$.

EXAMPLE For the alphabet of parentheses, $\Sigma = \{\texttt{)}, \texttt{(}\}$, the set of strings of balanced parentheses is a language. Three members are the strings `(())`, `(()())`, and `(()(()))`. Nonmembers are the strings `)(`, `(()`, and `())(()`.

DEFINITION (OPERATIONS ON LANGUAGES) The concatenation of languages, $\mathcal{L}_0 \frown \mathcal{L}_1$ or $\mathcal{L}_0\mathcal{L}_1$, is the language of concatenations, $\{\sigma_0 \frown \sigma_1 \mid \sigma_0 \in \mathcal{L}_0 \text{ and } \sigma_1 \in \mathcal{L}_1\}$.

For any language $\mathcal{L}$, the power $\mathcal{L}^k$ is the language consisting of the concatenation of k-many members, $\mathcal{L}^k = \{\sigma_0 \frown \cdots \frown \sigma_{k-1} \mid \sigma_i \in \mathcal{L}\}$ when $k > 0$. We take $\mathcal{L}^0 = \{\varepsilon\}$. The Kleene star of a language $\mathcal{L}^*$ is the language consisting of the concatenation of any number of strings.

$$\mathcal{L}^* = \{\sigma_0 \frown \cdots \frown \sigma_{k-1} \mid k \in \mathbb{N} \text{ and } \sigma_0, \ldots, \sigma_{k-1} \in \mathcal{L}\} = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \cdots$$

This includes the concatenation of 0-many strings, so $\varepsilon \in \mathcal{L}^*$ even if $\mathcal{L} = \varnothing$.

The reversal of a language is the language of reversals, $\mathcal{L}^{\mathsf{R}} = \{\sigma^{\mathsf{R}} \mid \sigma \in \mathcal{L}\}$.

Deciding a language vs recognizing a language

We have already defined that a machine decides a language if it computes whether or not a given input is a member of that language.

For the other, suppose that the language is computably enumerable but not computable. Then there is a machine that determines whether a given input is a member of the language but it is not able to determine whether the input is not in the language. For instance, there is a Turing machine that, given input e, can determine whether $e \in K$ but no machine can determine whether $e \notin K$.

We will say that a machine recognizes (or accepts, or semidecides) a language when, given an input, the machine computes in a finite time whether the input is in the language, and if it is not in the language then the machine will never incorrectly report that it is. (The machine may determine that it is not, or it may simply fail to report a conclusion such as by failing to halt.)

EXAMPLE A Turing machine can decide the language of perfect squares, $S = \{ 1^{(n^2)} \mid n \in \mathbb{N} \}$. (Technically, it decides the language of strings representing squares, but we will not stick to the distinction.)

EXAMPLE Consider the set S of indices e such that $\mathcal{P}_e$ halts on input 3. A Turing machine can recognize S just by running the $\mathcal{P}_e$ on input 3 and waiting for it to halt. But no Turing machine decides that set, because we have shown that K reduces to S and so if we had a Turing machine to decide membership in S then we could use it to decide membership in K.)

Grammars

Introduction

The definition of language allows $\mathcal{L}$ to be any subset of Σ^* at all. But languages in practice usually are constructed according to patterns, to lists of rules.

An example showing that there are rules is that, “When nine hundred years old you reach, look as good you will not” may fit the rules wherever Yoda was raised, but it does not fit those for English. Another example is that the C language consists of programs that follow rules decreed by the standardization committee.

Introduction

The definition of language allows $\mathcal{L}$ to be any subset of Σ^* at all. But languages in practice usually are constructed according to patterns, to lists of rules.

An example showing that there are rules is that, “When nine hundred years old you reach, look as good you will not” may fit the rules wherever Yoda was raised, but it does not fit those for English. Another example is that the C language consists of programs that follow rules decreed by the standardization committee.

This is a subset of the rules for English:

- a sentence can be made from a noun phrase followed by a verb phrase,
- a noun phrase can be made from an article followed by a noun,
- a noun phrase can also be made from an article then an adjective then a noun,
- a verb phrase can be made with a verb followed by a noun phrase,
- one article is ‘the’,
- one adjective is ‘young’,
- one verb is ‘caught’,
- two nouns are ‘man’ and ‘ball’.

We will write the above rules in this form.

⟨*sentence*⟩ → ⟨*noun phrase*⟩ ⟨*verb phrase*⟩
⟨*noun phrase*⟩ → ⟨*article*⟩ ⟨*noun*⟩
⟨*noun phrase*⟩ → ⟨*article*⟩ ⟨*adjective*⟩ ⟨*noun*⟩
⟨*verb phrase*⟩ → ⟨*verb*⟩ ⟨*noun phrase*⟩
⟨*article*⟩ → `the`
⟨*adjective*⟩ → `young`
⟨*verb*⟩ → `caught`
⟨*noun*⟩ → `man` | `ball`

Each line is a production or rewrite rule. Each has one arrow, →. To the left of each arrow is a head and to the right is a body or expansion. Sometimes two rules have the same head, as with ⟨*noun phrase*⟩. There are also two rules for ⟨*noun*⟩ but we have abbreviated by combining the bodies using the '|' pipe symbol.

The rules use two different components. The ones written in typewriter type, such as young, are from the alphabet Σ of the language. These are terminals. The ones with angle brackets and italics, such as ⟨*article*⟩, are nonterminals. These are like variables and are used for intermediate steps.

The two symbols '→' and '|' are neither terminals nor nonterminals. They are metacharacters, part of the syntax of the rules themselves.

Derivation

The rewrite rules govern the derivation of strings in the language. Under the grammar for English every derivation starts with ⟨*sentence*⟩. During a derivation, intermediate strings contain a mix of nonterminals and terminals. In our grammars every rule has a head with a single nonterminal. So to get the next string pick a nonterminal in the present string and substitute a matching body.

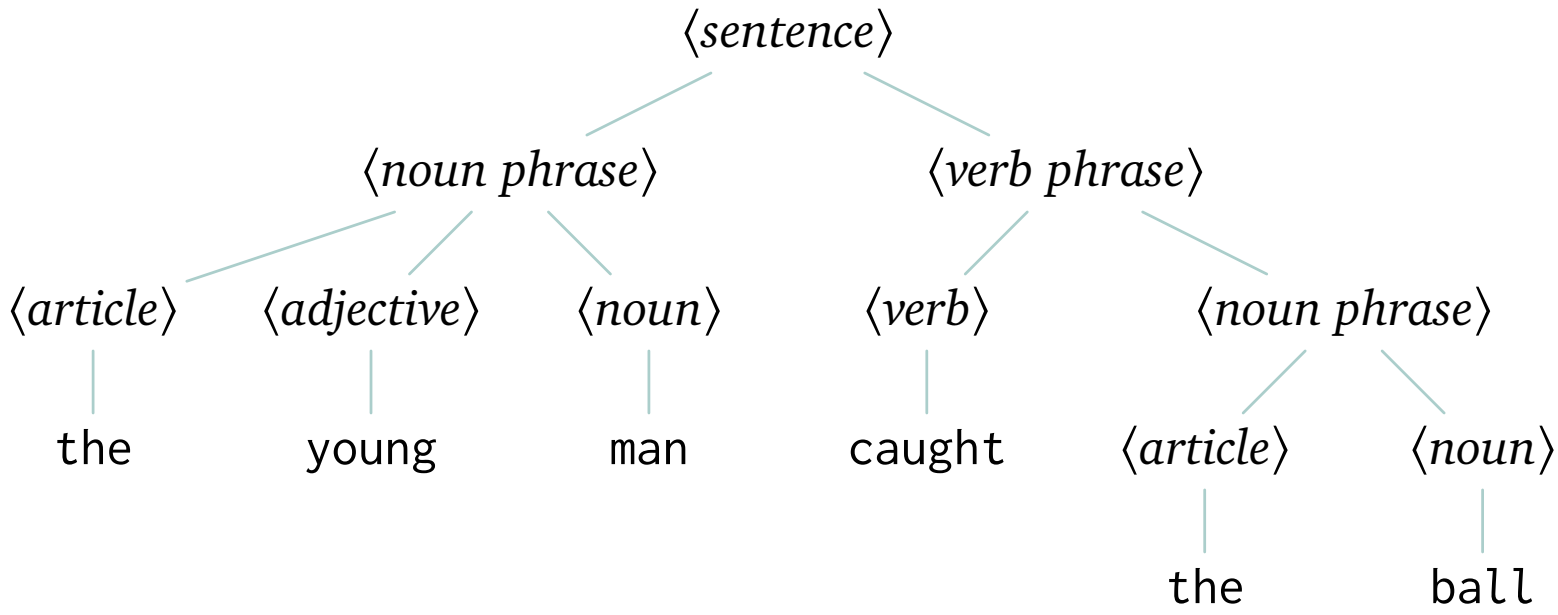
$$
\begin{aligned}
\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun phrase} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \langle \textit{article} \rangle \ \langle \textit{adjective} \rangle \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the} \ \langle \textit{adjective} \rangle \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young} \ \langle \textit{noun} \rangle \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \ \langle \textit{verb phrase} \rangle \\
&\Rightarrow \texttt{the young man} \ \langle \textit{verb} \rangle \ \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \ \langle \textit{noun phrase} \rangle \\
&\Rightarrow \texttt{the young man caught} \ \langle \textit{article} \rangle \ \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the} \ \langle \textit{noun} \rangle \\
&\Rightarrow \texttt{the young man caught the ball}
\end{aligned}
$$

Note that the single line arrow $\rightarrow$ is for rules while the double line arrow $\Rightarrow$ is for derivations.

The derivation above always substitutes for the leftmost nonterminal, so it is a leftmost derivation. However, in general we could substitute for any nonterminal.

Derivation tree

The derivation tree or parse tree is an alternative representation.



Grammar

In this course we will use the definition of grammar below. It is not the most general definition possible; for instance, we could allow the head to have more than just a single nonterminal. Our sticking to this more restrictive form is nonstandard so that you may well see other authors using other definitions, but this is the most general form that we shall need.

DEFINITION A context-free grammar is a four-tuple $\mathcal{G} = \langle \Sigma, N, S, P \rangle$. The set Σ is an alphabet, whose elements are the terminal symbols, and the elements of the set N are the nonterminals or syntactic categories. (We assume that Σ and N are disjoint and that neither contains metacharacters.) The symbol $S \in N$ is the start symbol. Finally, P is a set of productions or rewrite rules.

EXAMPLE Let the alphabet be $\Sigma = \{ \mathsf{a}, \mathsf{b} \}$, let the set of nonterminals be $N = \{ \langle \mathit{start} \rangle, \langle \mathit{left} \rangle, \langle \mathit{right} \rangle \}$. Here are five production rules making the set P.

$$\begin{aligned}
\langle \mathit{start} \rangle &\to \mathsf{a}\langle \mathit{left} \rangle \\
\langle \mathit{left} \rangle &\to \mathsf{a}\langle \mathit{left} \rangle \mid \mathsf{b}\langle \mathit{right} \rangle \\
\langle \mathit{right} \rangle &\to \mathsf{b}\langle \mathit{right} \rangle \mid \varepsilon
\end{aligned}$$

As to the start symbol, we take the convention that it is the head of the first rule. Here is a derivation using the production rules.

$$\langle \mathit{start} \rangle \Rightarrow \mathsf{a}\langle \mathit{left} \rangle \Rightarrow \mathsf{aa}\langle \mathit{left} \rangle \Rightarrow \mathsf{aab}\langle \mathit{right} \rangle \Rightarrow \mathsf{aabb}\langle \mathit{right} \rangle \Rightarrow \mathsf{aabb}\varepsilon = \mathsf{aabb}$$

EXAMPLE This is part of the grammar of C.

⟨*statement_list*⟩ → ⟨*statement*⟩ | ⟨*statement_list*⟩ ⟨*statement*⟩

⟨*expression_statement*⟩ → ; | ⟨*expression*⟩ ⟨*;*⟩

⟨*selection_statement*⟩ → `IF` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩
| `IF` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩ `ELSE` ⟨*statement*⟩
| `SWITCH` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩

⟨*iteration_statement*⟩ → `WHILE` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩
| `DO` ⟨*statement*⟩ `WHILE` `(` ⟨*expression*⟩ `)` `;`
| `FOR` `(` ⟨*expression_statement*⟩ ⟨*expression_statement*⟩ `)` ⟨*statement*⟩
| `FOR` `(` ⟨*expression_statement*⟩ ⟨*expression_statement*⟩ ⟨*expression*⟩ `)` ⟨*statement*⟩

(Here the capitalized terminals mean any string that capitalizes to that, so that `IF` means `If`, or `iF`, or `if`, or `IF`.)

Recursive grammars

The prior two example grammars are recursive. For instance in this rule,

$$\langle \textit{left} \rangle \;\longrightarrow\; \texttt{a}\langle \textit{left} \rangle \;\mid\; \texttt{b}\langle \textit{right} \rangle$$

$\langle \textit{left} \rangle$ can expand to an expression involving $\langle \textit{left} \rangle$. Similarly, in the C grammar $\langle \textit{statement\_list} \rangle$'s expansion involves itself.

$$\langle \textit{statement\_list} \rangle \;\longrightarrow\; \langle \textit{statement} \rangle \;\mid\; \langle \textit{statement\_list} \rangle\, \langle \textit{statement} \rangle$$

But we don't get stuck in an infinite regress. The question is not whether you could perversely keep expanding $\langle \textit{left} \rangle$ or $\langle \textit{statement\_list} \rangle$ forever. Instead, you are given a string such as aabb and the question is whether you can find a terminating derivation.

An abbreviation convention

For examples and exercises, often for nonterminals we use uppercase letters and for terminals we use other characters such as lowercase letters or digits.

EXAMPLE This grammar has two nonterminals, S and I. Its terminals are the parentheses, `(` and `)`, the arithmetic operators, `+` and `*`, and the digits.

S → `(` S `)` | S `+` S | S `*` S | I

I → `0` | `1` | `...` `9`

The derived strings are arithmetic expressions such as `1+2*(3+4)`.

Language of a grammar

DEFINITION The language derived from a grammar is the set of strings of terminals having derivations that begin with the start symbol.

EXAMPLE This grammar G

$$\begin{aligned} \mathrm{S} &\longrightarrow \mathrm{AB} \\ \mathrm{A} &\longrightarrow \mathrm{aA} \mid \mathrm{a} \\ \mathrm{B} &\longrightarrow \mathrm{bB} \mid \mathrm{b} \end{aligned}$$

lets you derive strings consisting of some a's followed by some b's.

$$\mathcal{L}(G) = \{\, \mathsf{a}^n \mathsf{b}^m \;\big|\; n, m \in \mathbb{N}^+ \,\}$$

EXAMPLE This grammar $\hat{G}$

$$\begin{aligned} \mathrm{S} &\longrightarrow \mathrm{AB} \\ \mathrm{A} &\longrightarrow \mathrm{a} \\ \mathrm{B} &\longrightarrow \mathrm{bbB} \mid \varepsilon \end{aligned}$$

lets you derive strings consisting of a single a, and then an even number of b's, including possibly zero many b's.

$$\mathcal{L}(\hat{G}) = \{\, \mathsf{ab}^{2m} \;\big|\; m \in \mathbb{N} \,\}$$

This is a kind of computation

Derivations involve pushing symbols around according to formal rules. So it fits with our idea of a mechanical computation.

For instance, this grammar

$$S \rightarrow 11S \mid \varepsilon$$

produces the language $\{1^{2n} \mid n \in \mathbb{N}\}$ of even numbers in unary notation.

In a similar way we can compute such things as the set of perfect squares. (As we remarked earlier, there are grammar types that are more general than the ones we've defined. With more general grammars we can compute more sets. An example is that the set of primes requires such a grammar. But we won't compute these; having made the point that derivation is a form of computation we will stop.)

A grammar can be ambiguious

This bit of C-like code shows nested `if` statements.

```
if ... then ... if ... then ... else ...
```

The `else` could be associated with either the first `if` or the second.

```
if ...
then ...
     if ...
     then ...
else ...
```

```
if ...
then ...
     if ...
     then ...
     else ...
```

This is a dangling else. An ambiguous grammar is one where some strings have two different leftmost derivations. The example below illustrates.

This grammar

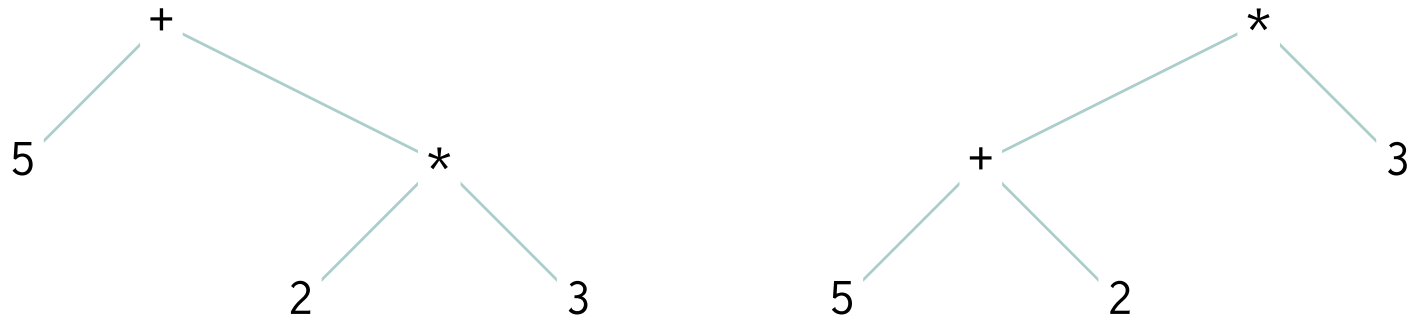
$$\begin{aligned}\langle expr\rangle \;&\longrightarrow\; \langle expr\rangle \texttt{+} \langle expr\rangle \\ &\mid \langle expr\rangle \texttt{*} \langle expr\rangle \\ &\mid \texttt{(}\ \langle expr\rangle\ \texttt{)} \;\mid \texttt{0} \mid \texttt{1} \mid \ldots \texttt{9}\end{aligned}$$

is ambiguous because `5+2*3` has two leftmost derivations.

$$\begin{aligned}\langle expr\rangle &\Rightarrow \langle expr\rangle\texttt{+}\langle expr\rangle \;\Rightarrow\; \texttt{5 + }\langle expr\rangle \\ &\Rightarrow \texttt{5 + }\langle expr\rangle \texttt{ * }\langle expr\rangle \;\Rightarrow\; \texttt{5 + 2 * }\langle expr\rangle \;\Rightarrow\; \texttt{5 + 2 * 3}\end{aligned}$$

$$\begin{aligned}\langle expr\rangle &\Rightarrow \langle expr\rangle \texttt{ * } \langle expr\rangle \;\Rightarrow\; \langle expr\rangle \texttt{ + } \langle expr\rangle \texttt{ * } \langle expr\rangle \\ &\Rightarrow \texttt{5 + }\langle expr\rangle \texttt{ * }\langle expr\rangle \;\Rightarrow\; \texttt{5 + 2 * }\langle expr\rangle \;\Rightarrow\; \texttt{5 + 2 * 3}\end{aligned}$$

Here are the two associated parse trees.



The left one evaluates to $5 + 6 = 11$, while the right one gives $7 \cdot 3 = 21$.

BNF

Backus-Naur notation

Many programming languages, protocols, or formats are specified using Backus-Naur form. Every rule has the structure: ⟨*name*⟩ ::= ⟨*expansion*⟩.

For instance, this is part of a BNF grammar of arithmetic expressions.

```
<expr> ::= <term> "+" <expr>
         |  <term>

<term> ::= <factor> "*" <term>
         |  <factor>

<factor> ::= "(" <expr> ")"
           |  <const>

<const> ::= integer
```

As a first point, it is a more type-able version of this.

⟨*expr*⟩ → ⟨*term*⟩ + ⟨*expr*⟩ | ⟨*term*⟩

⟨*term*⟩ → ⟨*factor*⟩ * ⟨*term*⟩ | ⟨*factor*⟩

⟨*factor*⟩ → (⟨*expr*⟩) | ⟨*const*⟩

⟨*const*⟩ → ⟨*integer*⟩

But there are also a number of extensions.

Common extensions to BNF

Repetition To show that an element may be repeated zero or more times, use curly braces. An example is that this describes a comma-separated argument list.

⟨*args*⟩ ::= ⟨*arg*⟩ { , ⟨*arg*⟩ }

To denote zero or more times you may also see a Kleene star, '∗'.

⟨*trailing-space*⟩ ::= ⟨*whitespace*⟩*

To denote one or more times you may see a plus, '+'.

⟨*natural*⟩ ::= ⟨*digit*⟩+

For zero or one repetitions, you may see square brackets.

⟨*int*⟩ ::= [+ | -] ⟨*natural*⟩

Common extensions to BNF

Repetition To show that an element may be repeated zero or more times, use curly braces. An example is that this describes a comma-separated argument list.

⟨*args*⟩ ::= ⟨*arg*⟩ { , ⟨*arg*⟩ }

To denote zero or more times you may also see a Kleene star, '∗'.

⟨*trailing-space*⟩ ::= ⟨*whitespace*⟩*

To denote one or more times you may see a plus, '+'.

⟨*natural*⟩ ::= ⟨*digit*⟩+

For zero or one repetitions, you may see square brackets.

⟨*int*⟩ ::= [+ | −] ⟨*natural*⟩

Grouping Use parentheses. This shows the form of addition or subtraction.

⟨*expr*⟩ ::= ⟨*term*⟩ (+ | −) ⟨*expr*⟩

The vertical bar for 'or' and the parentheses are metacharacters.

Common extensions to BNF

Repetition To show that an element may be repeated zero or more times, use curly braces. An example is that this describes a comma-separated argument list.

⟨*args*⟩ ::= ⟨*arg*⟩ { , ⟨*arg*⟩ }

To denote zero or more times you may also see a Kleene star, '∗'.

⟨*trailing-space*⟩ ::= ⟨*whitespace*⟩∗

To denote one or more times you may see a plus, '+'.

⟨*natural*⟩ ::= ⟨*digit*⟩+

For zero or one repetitions, you may see square brackets.

⟨*int*⟩ ::= [+ | −] ⟨*natural*⟩

Grouping Use parentheses. This shows the form of addition or subtraction.

⟨*expr*⟩ ::= ⟨*term*⟩ (+ | −) ⟨*expr*⟩

The vertical bar for 'or' and the parentheses are metacharacters.

Concatenation Juxtaposition is invisible so occasionally you see a comma ',' to denote concatenation.

Here is part of a grammar taken from RFC 5322. Note the quoted terminals.

```
date-time       =   [ day-of-week "," ] date time [CFWS]

day-of-week     =   ([FWS] day-name) / obs-day-of-week

day-name        =   "Mon" / "Tue" / "Wed" / "Thu" /
                "Fri" / "Sat" / "Sun"

date            =   day month year

day             =   ([FWS] 1*2DIGIT FWS) / obs-day

month           =   "Jan" / "Feb" / "Mar" / "Apr" /
            "May" / "Jun" / "Jul" / "Aug" /
            "Sep" / "Oct" / "Nov" / "Dec"

year            =   (FWS 4*DIGIT FWS) / obs-year

time            =   time-of-day zone

time-of-day     =   hour ":" minute [ ":" second ]

hour            =   2DIGIT / obs-hour

minute          =   2DIGIT / obs-minute

second          =   2DIGIT / obs-second

zone            =   (FWS ( "+" / "-" ) 4DIGIT) / obs-zone
```

Python's floating point

This is the grammar that the Python documentation uses for floating point numbers.

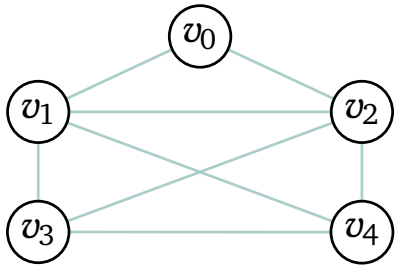
```
floatnumber   ::=  pointfloat | exponentfloat

pointfloat    ::=  [intpart] fraction | intpart "."

exponentfloat ::=  (intpart | pointfloat) exponent

intpart       ::=  digit+

fraction      ::=  "." digit+

exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

Graphs

Definition

Definition A simple graph is an ordered pair $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ where $\mathcal{N}$ is a finite set of vertices or nodes and $\mathcal{E}$ is a set of edges. Each edge is a set of two distinct vertices; these vertices are adjacent or neighbors.

Example This simple graph has five vertices $\mathcal{V} = \{v_0, \ldots, v_4\}$ and eight edges.



$$\mathcal{E} = \{\{v_0, v_1\}, \{v_0, v_2\}, \ \ldots \ \{v_3, v_4\}\}$$

Instead of writing $e = \{v, \hat{v}\}$ we often write $e = v\hat{v}$. Since edges are sets and sets are unordered we could write the same edge as $e = \hat{v}v$.

There are many variations of that definition, for modeling different circumstances. We could allow some vertices to connect to themselves, forming a loop. Another variant is a multigraph, which allows two vertices to share more than one edge.

Still another is a weighted graph, which gives each edge a real number weight, perhaps signifying the distance or the cost in money or time to traverse that edge.

A very common variation is a directed graph or digraph, where edges have a direction as in a road map that includes one-way streets. If an edge is directed from v to $\hat{v}$ then we can write it as $v\hat{v}$ but not in the other order.

Some important variations involve whether the graph has cycles. A cycle is a closed path around the graph; see the complete definition just below. A tree is an undirected connected graph with no cycles. A directed acyclic graph or DAG is a directed graph with no directed cycles.

A very common variation is a directed graph or digraph, where edges have a direction as in a road map that includes one-way streets. If an edge is directed from v to $\hat{v}$ then we can write it as $v\hat{v}$ but not in the other order.

Some important variations involve whether the graph has cycles. A cycle is a closed path around the graph; see the complete definition just below. A tree is an undirected connected graph with no cycles. A directed acyclic graph or DAG is a directed graph with no directed cycles.

DEFINITION Where $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ is a graph, a subgraph $\hat{\mathcal{G}} = \langle \hat{\mathcal{N}}, \hat{\mathcal{E}} \rangle$ satisfies $\hat{\mathcal{N}} \subseteq \mathcal{N}$ and $\hat{\mathcal{E}} \subseteq \mathcal{E}$. A subgraph with every possible edge, that is such that $v_i, v_j \in \hat{\mathcal{N}}$ and $e = v_i v_j \in \mathcal{E}$ implies that $e \in \hat{\mathcal{E}}$ also, is an induced subgraph.

Graph traversal

DEFINITION Two graph edges are adjacent if they share a vertex, so that they have the form $e_0 = uv$ and $e_1 = vw$. A walk is a sequence of adjacent edges $\langle v_0v_1,\ v_1v_2,\ \ldots\ v_{n-1}v_n\rangle$. Its length is the number of edges, n. If the initial vertex v_0 equals the final vertex v_n then the walk is closed, otherwise it is open. A trail is a walk where no edge occurs twice. A circuit is a closed trail. A path is a walk with no repeated edges or vertices, except that it may be closed and so the first and last vertices are equal. A closed path with at least one edge is a cycle.

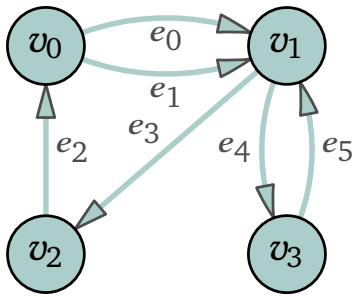
DEFINITION If a circuit contains all of a graph's edges then it is an Euler circuit. If it contains all of the vertices then it is a Hamiltonian circuit.

Graph representation

We can represent a graph in a computer with reasonable efficiency.

DEFINITION For a graph $\mathcal{G}$, the adjacency matrix $\mathcal{M}(\mathcal{G})$ representing the graph has i, j entries equal to the number of edges from v_i to v_j.

EXAMPLE This is the matrix for the graph.



$$\mathcal{M} = \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

LEMMA Let the matrix $\mathcal{M}(\mathcal{G})$ represent the graph $\mathcal{G}$. Then in its matrix multiplicative n-th power the i, j entry is the number of paths of length n from vertex v_i to vertex v_j.

EXAMPLE We can go from v_0 to v_1 in three steps in two different ways, via $e_0e_4e_5$ and via $e_1e_4e_5$.

$$\mathcal{M}^3 = \begin{pmatrix} 2 & 2 & 0 & 0 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

There is no way from v_3 to v_2 in three steps.

Graph coloring

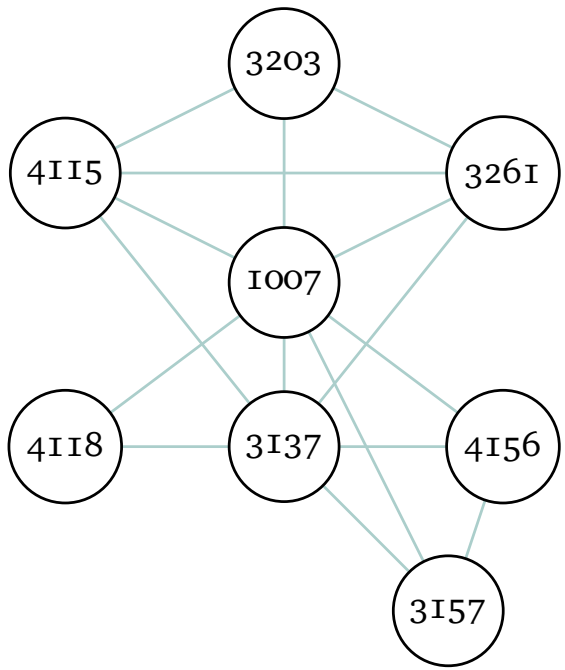
Definition A k-coloring of a graph, for $k \in \mathbb{N}$, is a partition of vertices into k-many classes such that no two adjacent vertices are in the same class.

Example We want to schedule final exams for the CS courses 1007, 3137, 3157, 3203, 3261, 4115, 4118, 4156. The classes 1007 and 3137 have students in common and so do these others: 1007-3157, 3137-3157, 1007-3203, 1007-3261, 3137-3261, 3203-3261, 1007-4115, 3137-4115, 3203-4115, 3261-4115, 1007-4118, 3137-4118, 1007-4156, 3137-4156, 3157-4156. What is the minimum number of time slots that we must use?

Graph coloring

Definition A k-coloring of a graph, for $k \in \mathbb{N}$, is a partition of vertices into k-many classes such that no two adjacent vertices are in the same class.

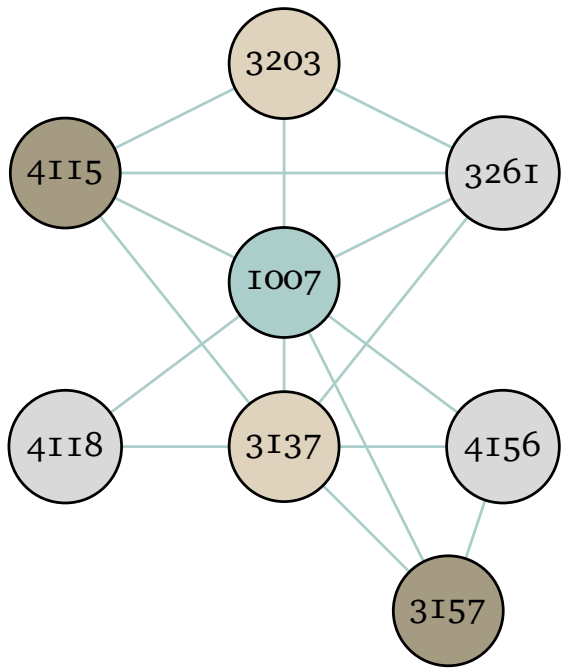
Example We want to schedule final exams for the CS courses 1007, 3137, 3157, 3203, 3261, 4115, 4118, 4156. The classes 1007 and 3137 have students in common and so do these others: 1007-3157, 3137-3157, 1007-3203, 1007-3261, 3137-3261, 3203-3261, 1007-4115, 3137-4115, 3203-4115, 3261-4115, 1007-4118, 3137-4118, 1007-4156, 3137-4156, 3157-4156. What is the minimum number of time slots that we must use?



Graph coloring

DEFINITION A k-coloring of a graph, for $k \in \mathbb{N}$, is a partition of vertices into k-many classes such that no two adjacent vertices are in the same class.

EXAMPLE We want to schedule final exams for the CS courses 1007, 3137, 3157, 3203, 3261, 4115, 4118, 4156. The classes 1007 and 3137 have students in common and so do these others: 1007-3157, 3137-3157, 1007-3203, 1007-3261, 3137-3261, 3203-3261, 1007-4115, 3137-4115, 3203-4115, 3261-4115, 1007-4118, 3137-4118, 1007-4156, 3137-4156, 3157-4156. What is the minimum number of time slots that we must use?



Graph isomorphism

We have a way to express that two graphs are, while not equal, essentially the same.

DEFINITION Two graphs $\mathcal{G}$ and $\hat{\mathcal{G}}$ are isomorphic if there is a one-to-one and onto map $f : \mathcal{N} \to \hat{\mathcal{N}}$ such that $\mathcal{G}$ has an edge $\{v_i, v_j\} \in \mathcal{E}$ if and only if $\hat{\mathcal{G}}$ has the associated edge $\{f(v_i), f(v_j)\} \in \hat{\mathcal{E}}$.

Showing that graphs are not isomorphic usually entails finding some graph-theoretic way in which they differ. A common and useful such property is to consider the degree of a vertex, the total number of edges touching that vertex with the proviso that a loop from the vertex to itself counts as two. The degree sequence of a graph is the non-increasing sequence of its vertex degrees.

If graphs are isomorphic then associated vertices have the same degree. Thus graphs with different degree sequences are not isomorphic. If the the degree sequences are equal then they help us construct an isomorphisms, if there is one, however there are graphs with the same degree sequence that are not isomorphic.