1. Language and Syntax

Emil Sekerinski, **McMaster University, January 2023**

Language and Grammar

Every language is based on a *vocabulary*. Its elements are called *words* or *symbols* whose structure is of no further interest. The *syntax* determines which sequences of words, called *sentences*, belong to the language.

language	symbols
English	`eats` , `Kevin` , `a` , `banana` , ...
Roman numerals	`I` , `V` , `X` , `L` , `C` , `D` , `M`
identifiers in programs	`A` , `B` , ..., `a` , `b` , ..., `0` , `1` , ..., `_`
arithmetic expressions	`dist` , `rot` , `24` , `+` , `-` , `×` , `/` , ...

Question: What are other non-spoken languages?

Answer:

- Chemical formulae, e.g H_3O^+ for hydronium.
- Musical scores, with vocabulary , ♭, ♮, ♯, ♩, ♪, ♫, ♬, etc.
- Morse code, with vocabulary "•" (short), "—" (long), " " (pause).

Formally, a vocabulary V is a finite, non-empty set of (atomic) symbols. The set V* of all *finite sequences* or *strings* over V consists of

- the empty string $ε$,
- any symbol $x ∈ V$,
- the *concatenation* $στ$ of strings $σ, τ ∈ V*$.

Example: if $V = \{a, b\}$, then
$V^+ = \{a, b, aa, ab, ba, bb, aaa, … \}$
$V* = \{ε, a, b, aa, ab, ba, bb, aaa, … \}$
The sentences of the language
$L = \{σaσ \mid σ ∈ V*\}$
are those sequences that contain at least one `a` .

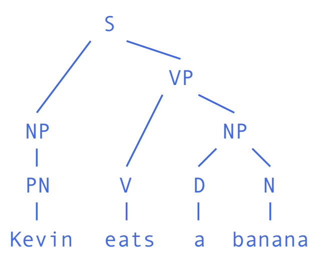
The empty sequence is both the left and right identity of concatenation and concatenation is associative, meaning that parenthesis can be left out. Formally for any $σ, τ, ω ∈ V*$:

- $σε = σ = εσ$
- $(στ)ω = σ(τω)$

The set of all *non-empty strings* over `V` is denoted by V^+ , formally $V^+ = V* - \{ε\}$. The *length* of string $σ$ is written as $|σ|$:

- $|ε| = 0$,
- $|x| = 1$ for any $x ∈ V$,
- $|στ| = |σ| + |τ|$ for any $σ, τ ∈ V*$.

A *grammar* not only determines unambiguously which sequences of words are sentences and which not but also provides sentences with a *structure*. The structure is instrumental in recognizing the *semantic* of a sentence, which is our ultimate goal.



The theory of formal languages originates in linguistics. A basic rule of English is that sentences (`S`) consists of a noun phrase (`NP`) followed by verb phrase (`VP`). A noun phrase is either a proper name (`PN`) or a determiner (`D`) followed by a noun (`N`). A verb phrase is either a verb (`V`) or a verb followed by a noun phrase. Determiners are `a` and `the` . The hierarchical composition of an English sentence by a *parse tree* is given to the right; below are the corresponding rules. Grammars of this form are called *generative* and the rules are called *productions*, as they determine how all sentences of a language are generated.

Formally, grammar $G = (T, N, P, S)$ is specified by

- a finite set `T` of *terminal symbols*,
- a finite set `N` of *nonterminal symbols*,
- a finite set `P` of *productions*,
- a symbol $S ∈ N$, the *start symbol*

where $N ∩ T = \{\}$ and $V = T ∪ N$ is its vocabulary. Productions are pairs of strings $σ ∈ V^+$, $τ ∈ V*$, written $σ → τ$.

```
S → NP VP
NP → PN
NP → D N
VP → V
VP → V NP
PN → Kevin
PN → Dave
D → a
D → the
```

Example. $G_0 = (T, N, P, S)$ with $T = \{Kevin, Dave, a, the, banana, apple, eats, runs\}$, $N = \{S, NP, VP, PN, D, N, V\}$, and the productions to the right is a grammar.

```
N → banana
N → apple
V → eats
V → runs
```

Given grammar $G = (T, N, P, S)$, sequence $\chi \in V^*$ is *directly derivable* from $\pi \in V^+$, written $\pi \Rightarrow \chi$, if there exist sequences σ, τ, μ, ν such that, $\pi = \mu\sigma\nu$, $\chi = \mu\tau\nu$, and $\sigma \to \tau \in P$.

If χ is derivable from π is n steps is written as $\pi \Rightarrow^n \chi$. Formally, relation $\Rightarrow^n$ is defined for $n \geq 0$ by:

- $\pi \Rightarrow^0 \pi$
- $\pi \Rightarrow^{n+1} \pi$ if $\pi \Rightarrow \rho$ and $\rho \Rightarrow^n \pi$ for some ρ

We write

- $\pi \Rightarrow^* \chi$ if χ is *derivable in zero or more steps* from π,
- $\pi \Rightarrow^+ \chi$ if χ is *derivable in one or more steps* from π.

```
S
⇒ NP VP
⇒ PN VP
⇒ Kevin VP
⇒ Kevin V NP
⇒ Kevin eats NP
⇒ Kevin eats D N
⇒ Kevin eats a N
⇒ Kevin eats a banana
```

Formally, $\Rightarrow^*$ is the transitive and reflexive closure of relation $\Rightarrow$ and $\Rightarrow^+$ is the transitive closure of $\Rightarrow$.

The derivation to the right allows to conclude that $S \Rightarrow^+ \text{Kevin eats a banana}$ with grammar G_0. More precisely, we can state $S \Rightarrow^8 \text{Kevin eats a banana}$.

The *language* $L(G)$ generated by grammar $G = (T, N, P, S)$ is the set of all sequences of terminal symbols which can be derived from the start symbol:

$$L(G) = \{\chi \in T^* \mid S \Rightarrow^+ \chi\}$$

Two grammars G, G' are *equivalent* if they generate the same language, $L(G) = L(G')$.

Example. Given $G_1 = (T, N, P, S)$, where $T = \{a, b, c, d\}$, $N = \{S, X\}$, $P = \{S \to aX, S \to bX, X \to c, X \to d\}$, the sequence ac is derivable from S, formally $S \Rightarrow^+ ac$,

$$S \Rightarrow aX \Rightarrow ac$$

as are ad, bc, bd. The language generated by G_1 is:

$$L(G_1) = \{ac, ad, bc, bd\}$$

Question. What are other equivalent grammars?

Answer.

- $G_1' = (T, N', P', S)$, where $N = \{S, X, Y\}$, $P = \{S \to XY, X \to a, X \to b, Y \to c, Y \to d\}$, is equivalent to G_1.
- Renaming the non-terminals also gives an equivalent grammar. In that sense, non-terminals "carry no meaning".
- Adding nonterminal X_1 and replacing $X \to a$ with $X \to X_1, X_1 \to a$ also gives an equivalent grammar. Repeating this, infinitely many equivalent grammars can be obtained.

Languages generated by a grammar can be *finite* or *infinite*. Infinite languages are expressed through recursion with a finite set of productions.

Example. Let $G_2 = (T, N, P, S)$, where $T = \{a\}$, $N = \{S\}$ and let the productions P be:

$$S \to \varepsilon$$
$$S \to aS$$

For a string σ, the term σ^n stands for σ repeated n times, formally $\sigma^0 = \varepsilon$ and $\sigma^{n+1} = \sigma\sigma^n$. For example, $\{a^n \mid n \geq 0\}$ is $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$.

Theorem. The language of G_2 is that of sequences over a of arbitrary length,

$$L(G_2) = \{a^n \mid n \geq 0\}$$

Proof. This is formally proved by inclusion in both directions. By definition of $L(G_2)$,

$$\{\chi \in T^* \mid S \Rightarrow^+ \chi\} \subseteq \{a^n \mid n \geq 0\}$$

means that for every $\chi \in T^*$ that is derivable from S there exists an $n \geq 0$ such that $\chi = a^n$. We show this by induction over the length of derivations from S.

- *Base.* A derivation of χ of length 1 from S can only derive $\chi = \varepsilon$ by the first production. As $\varepsilon = a^0$, the base case holds.
- *Step.* We need to show that each χ derivable from S in $n + 1$ steps, $S \Rightarrow^{n+1} \chi$ is a^i for some $i \geq 0$, under the hypothesis that each χ derivable from S in n steps, $S \Rightarrow^n \chi$ is a^i for some $i \geq 0$. If χ is derivable in $n+1$ steps, then $S \Rightarrow aS \Rightarrow^n \chi$ and χ is $a\omega$ for some ω. Since ω is derived from S in n steps, ω is a^i for some $i \geq 0$, hence $\chi = a\omega$ is a^i for some $i \geq 0$.

The inclusion in the other direction means that every a^n for $n \geq 0$ can be derived from S:

$$\{a^n \mid n \geq 0\} \subseteq \{\chi \in T^* \mid S \Rightarrow^+ \chi\}$$

We show this by induction over n.

- *Base.* For $n = 0$, obviously $a^0 = \varepsilon$ can be generated by the first production, $S \Rightarrow^+ \varepsilon$.
- *Step.* Suppose a^n can be generated, $S \Rightarrow^+ a^n$. We need to show that a^{n+1} can be generated as well. This follows from $S \Rightarrow aS \Rightarrow^+ aa^n = a^{n+1}$.

Thus we can conclude $L(G_2) = \{a^n \mid n \geq 0\}$.

Recursion also allows to express arbitrarily deep *nested structures*.

Example. Let $G_3 = (T, N, P, S)$, where $T = \{a, b, c\}$, $N = \{S\}$, and the productions P are:

```
S → b
S → aSc
```

The sequence $aabcc$ is derivable from S:

$$S \Rightarrow aSc \Rightarrow aaScc \Rightarrow aabcc$$

The generated language is:

$$L(G_3) = \{b, abc, aabcc, aaabccc, \ldots\} = \{a^n bc^n \mid n \geq 0\}$$

Chomsky Hierarchy

Languages can be classified according to restrictions on their grammar. The following classification is known as the *Chomsky Hierarchy* (Chomsky 1956). For grammar $G = (T, N, P, S)$, let $V = T \cup N$ be its vocabulary, and assume $a \in T$, $A, B \in N$, $\mu, \nu, \tau \in V^*$, $\sigma \in V^+$:

- A grammar is *context-sensitive* if productions are of the form

  $\mu A \nu \longrightarrow \mu \sigma \nu$

  Additionally, $S \rightarrow \varepsilon$ is allowed provided that S does not occur on the right hand side of another production

- A grammar is *context-free* if productions are of the form

  $A \longrightarrow \tau$

- A grammar is *regular* if productions are of the form

  $A \longrightarrow \varepsilon$ $A \longrightarrow a$ $A \longrightarrow aB$

Question. Which of the grammars G_0, G_1, G_2, G_3 are regular or context-free?

Answer.

- G_0 is not regular, but is context-free
- G_1 is regular (and therefore context-free)
- G_2 is regular (and therefore context-free)
- G_3 is not regular, but is context-free

Context-sensitive languages allow to express the subject-verb agreement with respect to singular vs. plural in natural languages.

Example. Consider the grammar to the right with terminals in lower case and nonterminals in upper case letters. Then

$$
\begin{aligned}
&S \to NP\ VP\\
&NP \to D\ N_s\\
&NP \to D\ N_p\\
&N_s\ VP \to N_s\ V_s\\
&N_p\ VP \to N_p\ V_p\\
&D \to \mathtt{the}\\
&N_s \to \mathtt{child}\\
&N_p \to \mathtt{children}\\
&V_s \to \mathtt{runs}\\
&V_p \to \mathtt{run}
\end{aligned}
$$

`the child runs`
`the children run`

are sentences but `the child run` is not.

Question. What is a derivation of `the child runs`?

Answer.

$$
\begin{aligned}
&\ \ S\\
&\Rightarrow NP\ VP\\
&\Rightarrow D\ N_s\ VP\\
&\Rightarrow D\ N_s\ V_s\\
&\Rightarrow \mathtt{the\ child}\ V_s\\
&\Rightarrow \mathtt{the\ child\ runs}
\end{aligned}
$$

We give some fundamental results from formal language theory. Regular grammars can express repetition, but not nesting:

Theorem. No regular grammar for $L(G_3)$ exists.

Example. Let $G_4 = (T, N, P, S)$, where $T = \{a, b, c\}$, $N = \{S, B\}$, and let the productions P be:

```
S → abc
S → aBSc
Ba → aB
Bb → bb
```

The grammar is not context-free. The language generated is:

$$L(G_4) = \{abc, aabbcc, aaabbbccc, …\} = \{a^n b^n c^n \mid n \geq 1\}$$

Question. What is a derivation of `aaabbbccc` in G_4? Explain how the grammar works!

Answer.

```
  S
⇒ aBSc
⇒ aBaBScc
⇒ aBaBabccc
⇒ aBaaBbccc
⇒ aaBaBbccc
⇒ aaaBBbccc
⇒ aaaBbbccc
⇒ aaabbbccc
```

The grammar works by first producing the same number of `a`, `B`, `c`, with all `c` in correct position at the end but `a` and `B` alternating. The the production `Ba → aB` moves all `a` to the left and all `B` to the middle. Once a `B` is in its correct position, it is converted to a `b`.

Theorem. No context-free grammar for $L(G_4)$ exists.

Grammar G_4 is not context-sensitive: `Ba → aB` does not match the form for context-sensitive productions. However, grammar G_4' with the same terminals, additional nonterminal `X`, and following productions is context-sensitive and equivalent; it uses three productions to achieve `BA → AB` and adds production `A → a`:

```
S → Abc
S → ABSc
BA → BX
BX → AX
AX → AB
Bb → bb
A → a
```

Question. Argue that G_4' is context-sensitive. What is a derivation of `aabbcc` in G_4'?

Answer. The production `BA → BX` replaces `A` by `X` in left context `B`, so matches `μAν → μσν` with `μ`, `A`, `ν`, `σ` being `B`, `A`, `ε`, `X`. The production `BX → AX` replaces `B` with `Y` in right context `X`, so matches `μAν → μσν` with `μ`, `A`, `ν`, `σ` being `ε`, `B`, `X`, `Y`. The other productions are similar.

```
  S
⇒ ABSc
⇒ ABAbcc
⇒ ABXbcc
⇒ AAXbcc
⇒ AABbcc
⇒ AAbbcc
⇒ Aabbcc
⇒ aabbcc
```

Example. Let `G₅ = (T, N, P, S)`, where `T = {a, b}`, `N = {A, B, S}`, and productions `P` are:

`S → aAS`
`S → bBS`
`Aa → aA`
`Ab → bA`
`Ba → aB`
`Bb → bB`
`AS → Sa`
`BS → Sb`
`S → ε`

The grammar is not context-free. The language generated is the *copy language*:

```
L(G₅) = {ww | w ∈ T*}
```

The first two productions produce an arbitrary sequence of pairs of `aA` and `bB` ending with `S`. The following four productions move all `A` and `B` to the right without "overtaking" each other. The final three productions convert `A` to `a` and `B` to `b` from right to left.

Question. What is a derivation of `abab` in `G₅`?

Answer.

```
  S
⇒ aAS
⇒ aAbBS
⇒ abABS
⇒ abASb
⇒ abSab
⇒ abab
```

Theorem. No context-free grammar for `L(G₅)` exists.

Languages generated by context-sensitive, context-free, and regular grammars are called *context-sensitive*, *context-free*, and *regular languages*, respectively.

Theorem. Every regular language is also context-free. Every context-free language is also context-sensitive.

Note that the inclusion does not quite hold for grammars, as `A → ε` is allowed in regular and context-free grammars, but not in context-sensitive grammars.

For brevity, we write

```
σ → τ₀ | τ₁ | …
```

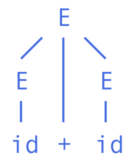
for the set of productions

```
σ → τ₀
σ → τ₁
…
```

Concrete and Abstract Syntax Trees

We continue with context-free languages. For those, the *parse tree* or *concrete syntax tree* is a visual representation of a derivation which abstracts from the order of independent applications of productions. In the example, `E` and `id` stand for expressions and identifiers of programs.

Example. Let G_6 `= (T, N, P, E)` where `T = {id, +}`, `N = {E}`, and the productions `P` are:

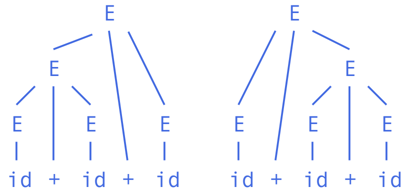



```
E → id | E + E
```

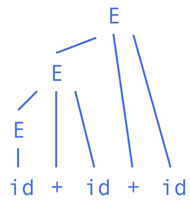
There are two derivations of `id + id`:

```
E ⇒ E + E ⇒ id + E ⇒ id + id
E ⇒ E + E ⇒ E + id ⇒ id + id
```

Continuing with G_6, there are two parse trees for `id + id + id`. A sentence with more than one parse trees is an *ambiguous sentence* and a grammar allowing that is an *ambiguous grammar*. Syntactically ambiguous sentences may have an ambiguous meaning. In natural languages this may be resolved through the context; in programming languages, syntactic ambiguity is generally avoided.

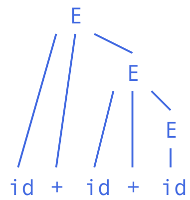


Changing the productions to a *left-recursive* form eliminates ambiguity and makes `+` associate to the left.




```
E → id | E + id
```

Changing the productions to a *right-recursive* form eliminates ambiguity and makes `+` associate to the right.




```
E → id | id + E
```

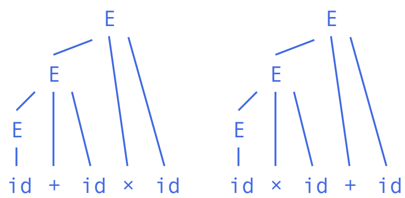
Question. For which operators in programming languages does associativity matter and for which not?

Answer.

- For integer division associativity matters.
- For integer addition associativity matters in bounded arithmetic (overflow is error) and saturating arithmetic (overflow results in maximal number).
- For integer addition associativity does not matter in modulo arithmetic, e.g. with word size, and with arbitrary precision.
- For bitwise `and` and bitwise `or`, associativity does not matter.
- For string concatenation, associativity does not matter.

The next example illustrates operator *precedence*.

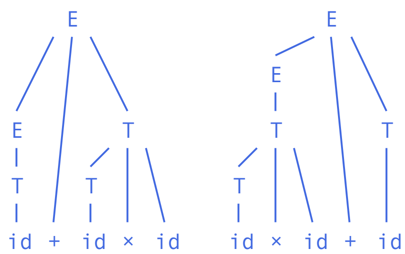
Example. Let G_7 `= (T, N, P, E)` where `T = {id, +, ×}`, `N = {E}`, and the productions `P` are:




```
E → id | E + id | E × id
```

In `id + id × id`, operator `+` binds tighter; in `id × id + id`, operator `×` binds tighter: `+` and `×` bind equally tight and associate to the left.

To have proper operator precedence, nonterminal `T` for terms is introduced and the productions are changed to:




```
E → T | E + T
T → id | T × id
```

To allow `+` to bind tighter than `×`, parenthesis are needed. For this, nonterminal `F` for factor is introduced.

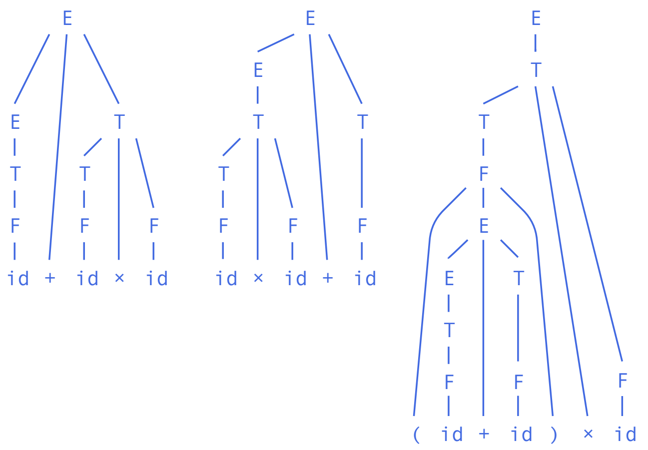
Example. Let G_8 `= (T, N, P, E)` where `T = {id, +, ×, (, )}`, `N = {E, T, F}`, and the productions `P` are:

```
E → T | E + T
```

```
T → F | T × F
F → id | ( E )
```

Question. What are the parse trees for `id + id × id`, for `id × id + id`, and for `(id + id) × id`?

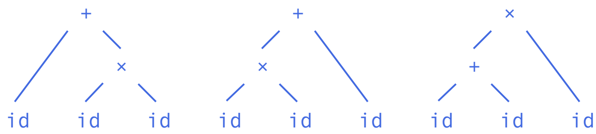
Answer.



A *structural tree* or *abstract syntax tree* is a simplified parse trees with only the relevant structure information:

- Productions whose sole purpose is to define precedence (like bracketing) are left out.
- Chains of derivations are left out.
- Nodes are labelled with the construct in question rather than a nonterminal.

For example, for `id + id × id`, for `id × id + id`, and for `(id + id) × id`:



A parse trees are also called a *concrete syntax tree*.

Backus-Naur Form

Context-free grammars are more conveniently written in *Backus-Naur Form* (*BNF*):

- The left-hand side of the first production is the start symbol.
- Terminals are enclosed in `'quotes'` all other symbols are nonterminals.
- Productions for the same nonterminal are grouped into one, separated by `|`.
- The empty string `ε` is written as `''`.

For example, here is BNF grammar for expression like `– 3 × a + b`

```
expression → term | '+' term | '–' term | expression '+' term | expression '–' term
term → factor | term '×' factor | term '/' factor
factor → number | identifier | '(' expression ')'
```

and one for statements like `if b then x := 3 else (x := y ; y := 5)`:

```
statement → assignment | compoundStatement | ifStatement | whileStatement
assignment → identifier ':=' expression
compoundStatement → '(' statementSequence ')'
statementSequence → statement | statementSequence ';' statement
ifStatement → 'if' expression 'then' statement | 'if' expression 'then' statement 'else'
statement
whileStatement → 'while' expression 'do' statement
```

Let us define BNF in BNF! The terminals are characters, written in quotes. The newline character is written as `\n` and the quote character itself as `\'`. We let `char` stand for an arbitrary character:

```
grammar  →  production | grammar '\n' production
production  →  identifier '→' expression
expression  →  term | expression '|' term
term  →  factor | term ' ' factor
factor  →  identifier | string
```

```
identifier  →  letter | identifier letter | identifier digit
letter  →  'A' | … | 'Z'
digit  →  '0' | … | '9'
string  →  '\'' characters '\''
characters  →  characters char | ''
```

Numerous variations of BNF exist. For example, the grammar of C uses different fonts for terminals and nonterminals, enumerates the terms of a production indented on subsequent lines, and uses `A_opt` if `A` is optional (Kernighan and Ritchie 1988). Formally, using `A_opt` amounts to adding a production `A_opt → A | ε`. Here is a simplified fragment:

```
statement:
        compound-statement
        expression-statement
        selection-statement
        iteration-statement
compound-statement:
        { statement-list_opt }
statement-list:
        statement
        statement-list statement
selection-statement:
        if ( expression ) statement
        if ( expression ) statement else statement
        switch ( expression ) statement
iteration-statement:
        while ( expression ) statement
        for ( expression_opt ; expression_opt ; expression_opt ) statement
```

EBNF is an extension of BNF that allows simple repetitions to be formulated more naturally and avoids an inflation of nonterminals:

- `(A)` allows precedence to be expressed. Formally, `(A)` stands for a new nonterminal `X` with the production `X → A` added.
- `[A]` stands for `A` optionally. Formally, `[A]` stands for a new nonterminal `X` with the production `X → A | ε` added.
- `{A}` stands for repeating `A` an arbitrary number of times. Formally, `{A}` stands for a new nonterminal `X` with the production `X → X A | ε` added.

For example, here is an EBNF grammar for expressions,

```
expression → [ '+' | '–' ] term { ( '+' | '–' ) term}
term → factor { ( '×' | '/' ) factor }
factor → number | identifier | '(' expression ')'
```

and one for statements:

```
statement → assignment | compoundStatement | ifStatement | whileStatement
assignment → identifier ':=' expression
compoundStatement → '(' statement { ';' statement } ')'
ifStatement → 'if' expression 'then' statement ['else' statement]
whileStatement → 'while' expression 'do' statement
```

Question. First, eliminate `(...)`, `[...]` in the expression grammar, then eliminate `{...}`. How can the grammar be made more readable?

For eliminating `(...)` and `[...]`, `{...}` we introduce `unaryop`, `addop`, and `multop`:

```
expression → unaryop term { addop term}
unaryop → '+' | '–' | ε
addop → '+' | '–'
term → factor { multop factor }
multop → '×' | '/'
factor → number | identifier | '(' expression ')'
```

For eliminating `{...}` in the production for `term`, that production can be replaced by:

```
term → factor morefactor
morefactor → morefactor multop factor | ε
```

The introduction of the nonterminal `morefactor` and the use of `ε` can be avoided here:

```
expression → unaryop primary
primary → term | primary addop term
unaryop → '+' | '−' | ε
addop → '+' | '−'
term → factor | term multop factor
multop → '×' | '/'
factor → number | identifier | '(' expression ')'
```

Let us define EBNF in EBNF!

```
grammar  →  production {'\n' production }
production  →  identifier '→' expression
expression  →  term { '|' term }
term  →  factor { ' ' factor }
factor  →  identifier | string | '(' expression ')' | '[' expression ']' | '{' expression '}'
identifier  →  letter { letter | digit }
letter  →  'A' | … | 'Z'
digit  →  '0' | … | '9'
string  →  '\'' { char } '\''
```

Sometimes `=` or `::=` is used instead of `→` and productions are terminated with a dot. For example, here is a fragment of the Go Grammar:

```
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .

Statement =
    Declaration | Assignment | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt .

Assignment = ExpressionList assign_op ExpressionList .
assign_op = [ add_op | mul_op ] "=" .

ExpressionList = Expression { "," Expression } .
```

Productions using `=` also called *syntactic equations*, however care has to be taken as `A = B` is not the same as `B = A`!

More variations of EBNF exist:

- Zero or more repetitions of `E` are also written as `E*`
- One or more repetitions of `E` are written as E^+, which stands `E E*`.
- An optional occurrence of `E` is also written as `E?`, which stands for `E | ε`.

Here is a fragment of the Python grammar in the language reference (which differs slightly from the grammar used by parsers). As in Python indentation of statements matters, this is expressed in the grammar by symbols that indicate indentation:

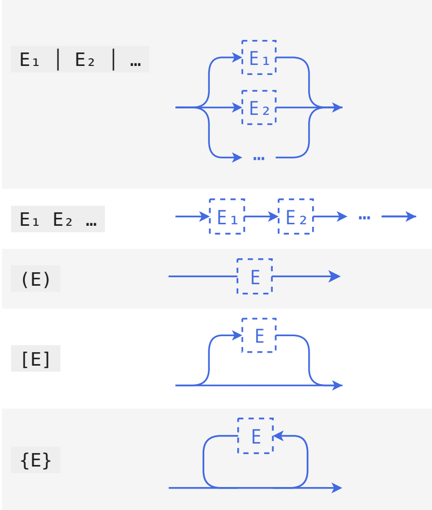
```
compound_stmt ::=  if_stmt
                   | while_stmt
                   | for_stmt
suite         ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement     ::=  stmt_list NEWLINE | compound_stmt
stmt_list     ::=  simple_stmt (";" simple_stmt)* [";"]

if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

EBNF is not only helpful for a compact definition of a grammar, but is also essential for the construction of a specific kind of recognizer. Our choice of EBNF is motivated by that.

Syntax Diagrams

An EBNF grammar can be equivalently represented by *syntax diagrams* (*railroad diagrams*). These are constructed recursively over the structure of EBNF grammars. Let `'a'` stand for a string (terminal), `A` for an identifier (nonterminal), and `E`, E_1, E_2, … for expressions (right-hand side of productions):

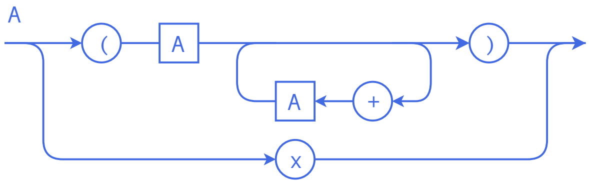
EBNF	syntax diagram
`A → E`	
`'a'`	
`A`	



For example, for

```
A → 'x' | '(' A { '+' A } ')'
```

the syntax diagram is:



Question. What is the syntax diagram for EBNF?

Recognizers

A *recognizer* for a language is a program that takes as input a string and *accepts* it if the string is a sentence of the language or otherwise *rejects* it. For regular, context-free, and context-sensitive languages, *universal recognizers* exist, i.e. programs that given a grammar `G` and sentence `ω` return if `ω ∈ L(G)`. For an unrestricted grammar `G`, in general `ω ∈ L(G)` is undecidable.

For context-sensitive grammar `G = (T, N, P, S)`, a universal recognizers can be constructed by generating all derivations of length `1`, length `2`, etc. from the start symbol and keeping a set, `d`, of the derived strings. New strings are only added to `d` if they are not longer than `ω` as in context-sensitive grammars, derived strings cannot shrink. This terminates if either `ω ∈ d`, in which case `ω` is accepted, or no more strings can be added to `d`, i.e. all derived strings of length of `ω` have been explored:

```
algorithm
procedure derivable(S, P, ω): boolean
    d₀, d := {}, {S}
    while d₀ ≠ d do
        d₀ := d
        for π ∈ d₀ do
            for σ → τ ∈ P do
                for μ, ν where π = μσν do
                    χ := μτν
                    if χ = ω then return true
                    else if |χ| ≤ |ω| then d := d ∪ {χ}
    return false
```

This algorithm always terminates and the memory it uses is bounded. Since the set `d` may be very large, it is not a practical universal recognizer, but a constructive proof that membership in a context-sensitive language is decidable.

For implementing in Python, symbols are represented by characters, i.e. strings as Python strings. The method `s.find(t, i)` returns the index of the first occurrence of `t` in `s` starting at index `i`, or `-1` if no such occurrence exists:

In [ ]:

```python
def derivable(S, P, ω):
    # S: start symbol, a string, P: productions, a set of pairs of strings, ω: string
    d0, d = {}, {S} # set of strings
    while d != d0:
        d0 = d
        for (σ, τ) in P:
            for π in d0:
                i = π.find(σ, 0) #print('π, i', π, i)
                while i != - 1:
```

```
                    χ = π[0:i] + τ + π[i + len(σ):] #print('χ', χ)
                    if χ == ω: return True
                    elif len(χ) <= len(ω): d = d.union({χ})
                    i = π.find(σ, i + 1) #print('d, i', d, i)
    return False
```

In [ ]:
```
derivable('S', {('S', 'a'), ('S', 'Sb')}, 'abb') # Grammar: S → a | Sb
```

In [ ]:
```
derivable('S', {('S', 'a'), ('S', 'Sb')}, 'bb') # Grammar: S → a | Sb
```

Abbreviating `the`, `child`, `children`, `runs`, `run` by `t`, `c`, `C`, `r`, `R` and `NP`, `VP`, N_s V_s, N_p V_p by $\mathscr{N}$, $\mathscr{V}$, `n`, `v`, `N`, `V`, the previous productions are expressed as:

In [ ]:
```
P = {('S', '𝒩𝒱'), ('𝒩', 'Dn'), ('𝒩', 'DN'), ('n𝒱', 'nv'), ('N𝒱', 'NV'),
     ('D', 't'), ('n', 'c'), ('N', 'C'), ('v', 'r'), ('V', 'R')}
```

In [ ]:
```
derivable('S', P, 'tCR')
```

In [ ]:
```
derivable('S', P, 'tcR')
```

Historic Notes and Further Reading

The Backus-Naur Form was first proposed by John Backus and then adopted by Peter Naur for the definition of Algol-60. Donald Knuth suggested the name (Knuth 1964). EBNF was proposed by Niklaus Wirth (Wirth 1977).

The original motivation for the classification of grammars came from the study of natural languages. Following examples illustrate the potential use of regular, context-free, and context-sensitive languages (credit for examples: C. Chesi, Univ. of Siena)

- *Right recursion* (*tail recursion*) of the form ab^n:

  [the dog bit [the cat [that chased [the mouse [that ran]]]]]

- *Center embedding* (*true recursion*) of the form $a^n b^n$:

  [the mouse [(that) the cat [(that) the dog bit] chased] ran]

- *Cross-serial dependencies* (*identity recursion*) of the form `ww`:

  John, Mary, and David, are a widower, a widow, and a widower, respectively

There is an ongoing discussion on using regular, context-free, and context-sensitive languages for natural languages. The male-female correspondence of the last example can also be seen as a semantic issue rather than a syntactic issue. If one takes the limits of human comprehension into account the full generality of context-sensitive, context-free, and even regular languages is not needed. As a consequence, further classes of grammars have emerged, e.g. (Kallmeyer 2010).

Grammars can be used for translation of natural languages: first the input sentence is parsed according to the grammar of the source language and then a sentence is generated that satisfies the grammar of the target languages. However, some recent works demonstrates that neural networks can perform better than grammar-based translation (Wu et al. 2016), (Le and Schuster 2016).

On the other hand, Chomsky's Hierarchy had a profound impact on computing: for each class of languages equivalent recognizers for languages are known. Calling languages of unrestricted grammars *recursively enumerable*, we have:

type	language	recognizer
type 0	recursively enumerable	Turing machine
type 1	context-sensitive	linear bounded automaton
type 2	context-free	pushdown automaton
type 3	regular	finite state automaton

Regular and context-free languages are ubiquitous as recognizers for those can be constructed efficiently and are themselves in some sense efficient. The next chapters in these notes discuss their use for scanning and parsing.

Even the above examples show the difficulty of writing context-sensitive grammars. After Algol 60 introduced the use of context-free grammars for its syntax, with Algol 68 an attempt was made to go beyond context-free grammars by using a dedicated "two-level grammar" (Wijngaarden et al. 1976); that kind of grammar was not used for another language. Around the same time, Knuth proposed *attribute grammars* as a way of associating computation (which can be type-checking and translation) to recognition of a context-free language (Knuth 68). Since then it has become common to define a programming language with regular and context-free grammars and to use attribute grammars for compilation. Type systems, which can be thought of as context-sensitive grammars, are also used in the definition of some languages (Cardelli 1996).

The Pascal language and its successors Modula-2 and Oberon have compact EBNF grammars. The syntax diagrams of the Apple Pascal can fit on a poster. It used to be common that these were hanging on the walls next to the computers!

Bibliography

Cardelli, Luca. 1996. "Type Systems." *ACM Comput. Surv.* 28 (1): 263–64. https://doi.org/10.1145/234313.234418.

Chomsky, N. 1956. "Three Models for the Description of Language." *IRE Transactions on Information Theory* 2 (3): 113–24. https://doi.org/10.1109/TIT.1956.1056813.

Kallmeyer, Laura. 2010. *Parsing Beyond Context-Free Grammars*. Springer-Verlag Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14846-0.

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Prentice Hall Professional Technical Reference.

Knuth, Donald E. "Backus Normal Form vs. Backus Naur Form.", Letter to Editor, *Communications of the ACM*, vol. 7, no. 12, Dec. 1964, pp. 735–36. *Dec. 1964*, doi:10.1145/355588.365140.

Knuth, Donald E. 1968. "Semantics of Context-Free Languages." *Mathematical Systems Theory* 2 (2): 127–45. https://doi.org/10.1007/BF01692511.

Le, Quoc V., and Mike Schuster. 2016. "A Neural Network for Machine Translation, at Production Scale." *Google AI Blog* (blog). September 27, 2016. https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html.

Wijngaarden, A. van, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, C. H. Lindsey, M. Sintzoff, L. G. L. T. Meertens, and R. G. Fisker, eds. 1976. *Revised Report on the Algorithmic Language Algol 68*. Berlin Heidelberg: Springer-Verlag. //www.springer.com/gp/book/9783540075929.

Wirth, Niklaus. 1977. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?" *Communications of the ACM* 20 (11): 822–23. https://doi.org/10.1145/359863.359883.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation." *CoRR* abs/1609.08144. http://arxiv.org/abs/1609.08144.