2.1.1. Formal Languages (10.1)

2.1.2. Grammars and Derivations (10.2)
   Grammars (10.2.1.)
   Examples of Grammars and Strings (10.2.2.)
   Derivations (10.2.3.)
   Language Generation (10.2.4.)
   Designing Syntax using Grammars (10.2.5.)

2.1.3. Modularity and BNF notation (10.3)
   A simple modular grammar (10.3.1.)
   The import construct (10.3.2.)
   BNF Notation (10.3.3.)
### Alphabet

#### Definition

An **Alphabet** is a finite non-empty set $T$. We shall consider the elements of $T$ to be symbols.

#### Examples

- The alphabet of decimal digits is
  \[ T_{Digit} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

- The lower-case alphabet of the English language is
  \[ T_{English\_Lowercase\_Alphabet} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} \]
Alphabet

The lower-case Welsh alphabet is

\[ T_{\text{Welsh Lowercase Alphabet}} = \{a, b, c, ch, d, dd, e, f, ff, g, ng, h, i, l, ll, m, n, o, p, ph, r, rh, s, t, th, u, w, y\} \]

Notice: j, k, q, v, x, z are not elements of this alphabet.

Therefore “Taxi” written as “Taksi” in Welsh.

The Unicode alphabet has over 100,000 characters.

The Ascii alphabet has 128 characters, of which 33 are non-printing control characters.
String

Definition

A **string** or **word** over an alphabet $T$ is a finite sequence elements of $T$. The set of all strings or words over the alphabet $T$ is

$$T^* := \{ t_1 t_2 \cdots t_n \mid n \geq 0, t_1, t_2, \ldots, t_n \in T \}$$

- Note that one example is the **empty string**, which is represented by $\epsilon$.
- Many languages, e.g. Haskell, identify Strings with lists of characters.
2.1.1. Formal Languages (10.1)

String in Haskell

```haskell
a :: String
a = "Hello"

b :: String
b = ['H','e','l','l','o']

csetzer@cspcanton:~> ghci
...
Prelude> :load testString.hs
...
*Main> a
"Hello"
*Main> b
"Hello"
```
Examples of Strings

- 1984 and 2000 are strings over the alphabet $T_{Digit}$ of decimal digits.
- forwards and sdrawkcab are words over the lower-case English alphabet $T_{English-Lowercase-Alphabet}$.
Length of a String

Definition

The **length** of a string is given by the function

\[ | \cdot | : T^* \to \mathbb{N} \]

\( |w| \) is the number of symbols from the alphabet it contains.

- If we identify strings with \( \text{List}(T) \), it is the length of the corresponding list.
- **Examples**
  \[ |2000| = 4, \ |\text{forwards}| = 8, \ |\epsilon| = 0 \]
- **\( T^+ \)** is the set of non-empty strings:
  \[ T^+ = T^* \setminus \{\epsilon\} = \{t_1 t_2 \cdots t_n \mid n \geq 1, t_1, \ldots, t_n \in T\} \]
2.1.1. Formal Languages (10.1)

Concatenation

Definition

1. The **concatenation** of two strings $u = u_1 \cdots u_m$ and $v = v_1 \cdots v_n$ is the string

   $$uv = u_1, \ldots, u_m, v_1, \ldots, v_n$$

2. The **concatenation function** is the function

   $$\cdot : T^* \times T^* \rightarrow T^*$$

- For instance, if $u = 1984$ and $v = 2000$ then $uv = 19842000$.
- We have

   $$|uv| = |u| + |v|$$
Concatenation

**Definition**

We define $w^n := \underbrace{ww \cdots w}_{n \text{ times}}$.

For instance

$$\text{forwards}^3 = \text{forwardsforswardsforswards}$$
A **formal language** $L$ over an alphabet $T$ is a subset $L$ of the set of strings over $T$, i.e.

$$L \subseteq T^*$$

We usually say **language** for formal language.
Example 1

Let \( T_{a,b} = \{a, b\} \)

We can enumerate \( T_{a,b}^* \) in the following way

\[
T_{a,b}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots \}
\]

So we write first strings of length 0, then of length 1, then of length 2 etc.

There is only one string of length 0, namely \( \epsilon \).

The strings of length \( n + 1 \) are obtained by adding \( a \) in front of each string of length \( n \) and then do the same with \( b \).

Exercise Write a program which enumerates the strings of length \( n \) for given \( n \).
Here are a few examples of languages over $T_{a,b}$:

1. $L = \emptyset$.
2. $L = \{a, b\}^*$.
3. $L = \{\epsilon\}$.
4. $L = \{a\}$.
5. $L = \{a, b\}$.
6. $L = \{a^n \mid n \text{ is even}\}$.
7. $L = \{a^n b^n \mid n \geq 0\}$.
8. $L = \{a^n b^{n+1} \mid n \geq 0\}$.
9. $L = \{(ab)^n \mid n \geq 0\}$.
Example 2: URLs

Let

\[ T = \{ a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9, -, _, /, ., m, : \} \]

Let

\[ L = \{ w \in T^* \mid w \text{ is an http address} \} \]

\( L \) contains simple addresses such as

http://www.w3.org or http://www.swansea.ac.uk

but complex examples like

http://www.google.com/search?q=swansea+university+computer+science

are not in \( L \), since for example ? and + are not in \( T \).
Example 3: Infix Numbers

Let

\[ T_{Infix\_Arithmetic} = \{0, 1, +, \}, \]  

Expressions of type natural numbers are strings such as

- (0 + 1) + 1,
- (1 + 0).0, etc.
Example 4: Prefix Numbers

Let

\[ T_{\text{Prefix\_Arithmetic}} = \{ \text{zero, succ, add, mult}, , , (, ) \} \]

Expressions of type natural numbers are expressions such as

- \( \text{succ(succ(zero))} \),
- \( \text{add(zero, succ(succ(zero)))} \)

Let

\[ L_{\text{Prefix\_Arithmetic}} = \{ w \in T_{\text{Prefix\_Arithmetic}}^* \mid w \text{ is an arithmetic expression} \} \]
Most parser generators (or compiler-compiler) have two phases:

▶ In phase one the incoming stream of characters is grouped into simple tokens. This can be words like zero, succ above. Or even longer words considered as one entity, such as `<Identifier>`.

▶ In phase two the text is parsed using a grammar which refers to the tokens generated in the first phase.

▶ Token generation will make use of regular expressions, parsing in phase two will use restricted context free grammars.
  ▶ Regular expressions and context free grammars will be introduced later.
Designing Syntax Using Formal Languages

The syntax of a language is defined in two steps:

1. Choose an alphabet $T$.
2. Define the language $L \subseteq T^*$.
Definition

Let $L \subseteq T^*$ be a formal language over $T$. The recognition problem for $T$ is:

Given any $w \in T^*$ decide whether or not $w \in L$. 
2.1.1. Formal Languages (10.1)

2.1.2. Grammars and Derivations (10.2)
   - Grammars (10.2.1.)
   - Examples of Grammars and Strings (10.2.2.)
   - Derivations (10.2.3.)
   - Language Generation (10.2.4.)
   - Designing Syntax using Grammars (10.2.5.)

2.1.3. Modularity and BNF notation (10.3)
   - A simple modular grammar (10.3.1.)
   - The import construct (10.3.2.)
   - BNF Notation (10.3.3.)
A Grammar $G = (T, N, S, P)$ consists of

1. a finite set $T$ called the *alphabet* of *terminal symbols*,
2. a finite set $N$ of *non-terminal symbols* or *variable symbols* such that $T \cap N = \emptyset$,
3. a special non-terminal symbol $S \in N$ called the *start symbol*,
4. a finite set $P$ of substitution or rewrite rules, called *productions*, each of which has the form $u \rightarrow v$ where
   4.1 The left hand string $u \in (T \cup N)^+$ (esp. $u$ is non-empty),
   4.2 The right hand string $v \in (T \cup N)^*$.

Note that both left and right hand strings can contain both terminals and non-terminals.
Presentation of Grammars

We present a grammar as a 4-tuple $G = (T, N, S, P)$ and also use a displayed version:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$T$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$P$</td>
</tr>
</tbody>
</table>
A grammar $G$ defines a formal language $L(G) \subseteq T^*$. The elements of $L(G)$ are the set of strings we can obtain as follows:

- Start with start symbol $S$.
- Select a production such that the left hand string is a substring of what you derived so far.
- Replace this substring by the right hand string of the production.
- Once you have obtained a string consisting of non-terminals, stop.

We will first give some examples and then a formal definition of $L(G)$. 
Example 1: Consider the grammar

\[ G^{[01]^*1} = (\{0, 1\}, \{S\}, S, \{S \rightarrow 1, S \rightarrow 0S, S \rightarrow 1S\}) \]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{[01]^*1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>0, 1</td>
</tr>
<tr>
<td>nonterminals</td>
<td>S</td>
</tr>
<tr>
<td>start symbol</td>
<td>S</td>
</tr>
<tr>
<td>productions</td>
<td>( S \rightarrow 1, S \rightarrow 0S, S \rightarrow 1S )</td>
</tr>
</tbody>
</table>
Example 1 (Generation of Strings)

We assign numbers to the production rules:

Rule 1 $S \rightarrow 1$
Rule 2 $S \rightarrow 0S$
Rule 3 $S \rightarrow 1S$

A derivation of $1001 \in L(G^{[01]*1})$ is as follows:

\[
\begin{align*}
S & \Rightarrow 1S \quad \text{Rule 3} \\
& \Rightarrow 10S \quad \text{Rule 2} \\
& \Rightarrow 100S \quad \text{Rule 2} \\
& \Rightarrow 1001 \quad \text{Rule 1}
\end{align*}
\]

A derivation of $011 \in L(G^{[01]*1})$ is as follows:

\[
\begin{align*}
S & \Rightarrow 0S \quad \text{Rule 2} \\
& \Rightarrow 01S \quad \text{Rule 3} \\
& \Rightarrow 011 \quad \text{Rule 1}
\end{align*}
\]
Derivations of strings in $G\{01\}^*1$

[Diagram showing derivations of strings in $G\{01\}^*1$]
We can see that the elements of $L(G^{01*1})$ are the strings in $\{0, 1\}^*$ which end with a 1:

$$L(G^{01*1}) = \{w1 \mid w \in \{0, 1\}^*\}$$

Remark

$G^{01*1}$ is an example of a regular grammar (this notion will be introduced later).
Example 2 (Grammars)

Consider the grammar

\[ G^{ab^n} = (\{a, b\}, \{S\}, S, \{S \rightarrow ab, S \rightarrow aSb\}) \]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{ab^n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>a, b</td>
</tr>
<tr>
<td>nonterminals</td>
<td>S</td>
</tr>
<tr>
<td>start symbol</td>
<td>S</td>
</tr>
<tr>
<td>productions</td>
<td>( S \rightarrow ab, S \rightarrow aSb )</td>
</tr>
</tbody>
</table>
Derivations of strings in $G^{a^n b^n}$
We can see that the elements of $L(G^{a^n b^n})$ are the strings of the form $a^n b^n$:

$$L(G^{a^n b^n}) = \{ a^n b^n \mid n \geq 1 \}$$

Remark

$G^{a^n b^n}$ is an example of a context-free grammar (this notion will be introduced later).
Example 3 (Grammars)

Consider the grammar

\[ G^{a^n b^n c^n} = (\{a, b, c\}, \{S, B, C, H\}, S, \]
\[ \{ S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow HB, HB \rightarrow HC, \]
\[ HC \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc \}) \]

displayed as follows

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{a^n b^n c^n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>a, b, c</td>
</tr>
<tr>
<td>nonterminals</td>
<td>S, B, C, H</td>
</tr>
<tr>
<td>start symbol</td>
<td>S</td>
</tr>
</tbody>
</table>
| productions | \( S \rightarrow aSBC, S \rightarrow aBC, \)
|             | \( CB \rightarrow HB, HB \rightarrow HC, HC \rightarrow BC, \)
|             | \( aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc. \) |
Example Derivation

A derivation of $aabbcc$ in this grammar is as follows:

$$
S \Rightarrow aSBC \\
\Rightarrow aaBCBC \\
\Rightarrow aaBHBC \\
\Rightarrow aaBHCC \\
\Rightarrow aaBBCC \\
\Rightarrow aabBCC \\
\Rightarrow aabbCC \\
\Rightarrow aabbcc 
$$
We can see that the elements of \( L(G^{a^n b^n c^n}) \) are the strings of the form \( a^n b^n c^n \):

\[
L(G^{a^n b^n c^n}) = \{ a^n b^n c^n | n \geq 0 \}
\]

Remark

\( G^{a^n b^n} \) is an example of a context-sensitive grammar (this notion will be introduced later).
Definition

1. Let \( G = (T, N, S, P) \) be a grammar, \( w \in (T \cup N)^+ \) be a non-empty word, \( w' \in (T \cup N)^* \) be a possibly empty word. We say that \( w' \) is derived from \( w \) in one step, with notation

\[
 w \Rightarrow_G w'
\]

iff there exists a production \( u \rightarrow v \in P \) such that we can write

\[
 w = sut \text{ and } w' = svt
\]

for some \( s, t \in (T \cup N)^* \).
2. We say as well \( w' \) is immediately generated from \( w \) or \( w' \) is one-step generated from \( w \), or

\[
\begin{align*}
  \Rightarrow & \quad \Rightarrow
\end{align*}
\]

for \( w \Rightarrow_G w' \).

So \( w' \) is the result of replacing in \( w \) a substring, which is equal to the right hand string of a production by the left hand string of that production.
Definition $\Rightarrow^*$ (Cont.)

Definition

(a) Let $G = (T, N, S, P)$ be a grammar, $w \in (T \cup N)^+$ be a non-empty word, $w' \in (T \cup N)^*$ be a possibly empty word.

We say that $w'$ is derived from $w$, with notation $w \Rightarrow^*_G w'$ iff

- either $w = w'$
- or there exists a sequence of non-empty words $w_0, \ldots, w_{n-1} \in (T \cup N)^+$ and a final (possibly empty) word $w_n \in (T \cup N)^*$ s.t.
  - $w_0 = w$,
  - $w_n = w'$
  - for $1 \leq i \leq n - 1$ we have $w_i \Rightarrow_G w_{i+1}$. 
2. We say as well \( w' \) is \textit{generated from} \( w \) or \( w \Rightarrow w' \) for \( w \Rightarrow_G w' \).
Remark

The above definitions introduced relations

$$\Rightarrow_G \subseteq (T \cup N)^+ \times (T \cup N)^*$$
$$\Rightarrow^*_G \subseteq (T \cup N)^+ \times (T \cup N)^*$$

$$\Rightarrow^*_G$$ is the reflexive and transitive closure of $$\Rightarrow_G$$
Language Generation (10.2.4.)

Definition

Let $G = (T, N, S, P)$ be a grammar. The language generated by the grammar $G$, denoted by $L(G) \subseteq T^*$ is defined as the set of terminal strings generated from the start symbol, i.e.

$$L(G) := \{ w \in T^* \mid S \Rightarrow_G^* w \}$$

Remark

If a language is generated by a grammar, then there are infinitely many grammars which generate the same language.
Equivalence of Grammars

Definition

We say that two grammars $G_1$ and $G_2$ are equivalent iff

$$L(G_1) = L(G_2)$$
Example

We show that

\[ L(G^{a^n b^n}) = \{a^n b^n \mid n \geq 1\} \]

One can easily show that

\[ S \Rightarrow^* t \iff \exists n \geq 0(t = a^n S b^n \lor t = a^{n+1} b^{n+1}) \]

- “⇒” follows by induction on length of the derivation \( S \Rightarrow^* t \).
- “⇐”: show first the assertion for \( t = a^n S b^n \) by induction on \( n \). Then the assertion for \( t = a^{n+1} b^{n+1} \) follows as well.
Designing Syntax using Grammars (10.2.5.)

For programming languages $L$ it is usually difficult to find a grammar $G$ s.t. $L = L(G)$.

- One problem is the fact that variables usually need to be defined before being used.
  - We will later see that languages with such a property can usually not be defined by a context-free grammar.

- Without any restrictions on the grammar it is possible, but languages with unrestricted grammars are difficult to parse.

- Instead one defines first a grammar which reflects the basic structure of the language. Then one selects those strings which fulfil the additional criteria.
Designing Syntax using Grammars

In order to define a language $L$ one usually proceeds as follows:

**Step 1** Choose an alphabet s.t. $L \subseteq T^*$.

**Step 2** Choose a simple grammar $G$ with alphabet $T$ s.t.

$$L \subseteq L(G) \subseteq T^*$$

The grammar should give a good explanation of how the language is formed.

**Step 3** Define an algorithm which determines for $t \in L(G)$ whether $t \in L$ or not.
2.1.1. Formal Languages (10.1)

2.1.2. Grammars and Derivations (10.2)
   Grammars (10.2.1.)
   Examples of Grammars and Strings (10.2.2.)
   Derivations (10.2.3.)
   Language Generation (10.2.4.)
   Designing Syntax using Grammars (10.2.5.)

2.1.3. Modularity and BNF notation (10.3)
   A simple modular grammar (10.3.1.)
   The import construct (10.3.2.)
   BNF Notation (10.3.3.)
We introduce a grammar for a simple while language for computing natural numbers. It will have the following components:

- identifiers,
- natural numbers,
- arithmetic expressions,
- Boolean expressions,
- programs.
# Grammar for Identifiers

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{Identifier}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>a, b, …, z, A, B, …, Z</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Letter, Id</td>
</tr>
<tr>
<td>start symbol</td>
<td>Id</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$Id \rightarrow Letter$</td>
</tr>
<tr>
<td></td>
<td>$Id \rightarrow Letter \ Id$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow a$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow b$</td>
</tr>
<tr>
<td></td>
<td>$...$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow z$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow A$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow B$</td>
</tr>
<tr>
<td></td>
<td>$...$</td>
</tr>
<tr>
<td></td>
<td>$Letter \rightarrow Z$</td>
</tr>
</tbody>
</table>
Grammar for Numbers

\[
\begin{align*}
\text{grammar} & \quad G^{\text{Number}} \\
\text{terminals} & \quad 0, 1, \ldots, 9 \\
\text{nonterminals} & \quad \text{Number, Digit} \\
\text{start symbol} & \quad \text{Number} \\
\text{productions} & \quad \\
\text{Number} & \rightarrow \text{Digit} \\
\text{Number} & \rightarrow \text{Digit} \text{ Number} \\
\text{Digit} & \rightarrow 0 \\
\text{Digit} & \rightarrow 1 \\
& \quad \ldots \\
\text{Digit} & \rightarrow 9
\end{align*}
\]
The grammar for arithmetic expressions \( G^{\text{Arithmetic Expression}} \) will import \( G^{\text{Identifier}} \) and \( G^{\text{Number}} \).

This means the following:

- The terminals/nonterminals/productions of these grammars are added to the terminals/nonterminals/productions of \( G^{\text{Arithmetic Expression}} \).
Grammar for Arithmetic Expressions

grammar $G^{Arithmetic\_Expression}$
import $G^{Identifier}, G^{Number}$
terminals $+,-$
nonterminals $AExp, AOp$
start symbol $AExp$
productions

- $AExp \rightarrow Id$
- $AExp \rightarrow Number$
- $AExp \rightarrow AExp AOp AExp$
- $AOp \rightarrow +$
- $AOp \rightarrow -$
Grammar for Boolean Expressions

grammar $G_{Boolean \_Expression}$

import $G_{Arithmetic \_Expression}$

terminals true, false, not, and, or, $\equiv$, $<$

nonterminals $BExp$, $BOp1$, $BOp2$, $RelOp$

start symbol $BExp$

productions

$BExp \rightarrow BOp1 \ BExp$
$BExp \rightarrow BExp \langle BOp2 \rangle \ BExp$
$BExp \rightarrow AExp \ RelOp \ AExp$
$BExp \rightarrow true$
$BExp \rightarrow false$
$BOp1 \rightarrow not$
$BOp2 \rightarrow and$
$BOp2 \rightarrow or$
$RelOp \rightarrow \equiv$
$RelOp \rightarrow <$
Grammar for While Programs

**Grammar**

\[ G_{\text{while}} \]

**Import**

\[ G_{\text{Arithmetic Expression}}, G_{\text{Boolean Expression}} \]

**Terminals**

skip, if, then, else, fi, while, do, od, :=, ;

**Nonterminals**

Program

**Start Symbol**

Program

**Productions**

Program → skip
Program → Id := AExp
Program → Program ; Program
Program → if BExp then Program else Program fi
Program → while BExp do Program od
The Import Construct and Modular Grammars (10.3.2)

Definition

Let $H$ be a grammar

- **grammar** $H$
- **import** $G$
- **terminals** $T_H$
- **nonterminals** $N_H$
- **start symbol** $S_H$
- **productions** $P_H$

which imports grammar $G$
Definition (Importing Grammars, Cont)

Let the grammar imported by $H$ be defined as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>$G$</td>
</tr>
<tr>
<td>terminals</td>
<td>$T_G$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N_G$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S_G$</td>
</tr>
<tr>
<td>productions</td>
<td>$P_G$</td>
</tr>
</tbody>
</table>
Definition (Importing Grammars, Cont)

Then $H$ denotes the grammar $F$ which

- has start symbol $S_H$
- and as nonterminals/terminals/productions the union of the nonterminals/terminals/productions of $G$ and $H$.

This grammar $F$ is called the **flattened form** of $H$. 
The flattened form $F$ of $H$ is therefore defined as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>$F$</td>
</tr>
<tr>
<td>terminals</td>
<td>$T_G \cup T_H$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N_G \cup N_H$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S_H$</td>
</tr>
<tr>
<td>productions</td>
<td>$P_G \cup P_H$</td>
</tr>
</tbody>
</table>
Definition

The **Backus-Naur-Form** or **BNF** is the presentation of a grammar using the following conventions:

1. The terminal symbols are often written in bold font.
2. The non-terminal symbols are familiar terms for syntactic components enclosed in angle brackets, e.g. `<statement>`, `<expression>`, `<identifier>`. 
3. The start symbol is the non-terminal that is presented first.
4. The symbol ::= replaces and extends → by listing the productions possible for a non-terminal; alternative possibilities for the right hand sides of a particular production are separated by |.
Example 1

For instance the rules

\[
\langle BExp \rangle ::= \text{true} \\
\langle BExp \rangle ::= \text{false} \\
\langle BExp \rangle ::= \langle BOp1 \rangle \langle BExp \rangle \\
\langle BExp \rangle ::= \langle BExp \rangle \langle BOp2 \rangle \langle BExp \rangle \\
\langle BExp \rangle ::= \langle AExp \rangle \langle RelOp \rangle \langle AExp \rangle
\]

are condensed into one rule

\[
\langle BExp \rangle ::= \text{true} \mid \text{false} \mid \langle BOp1 \rangle \langle BExp \rangle \mid \langle BExp \rangle \langle BOp2 \rangle \langle BExp \rangle \mid \langle AExp \rangle \langle RelOp \rangle \langle AExp \rangle
\]
Example 2

**bnf**

**Letter**

**rules**

\[\langle\text{Letter}\rangle ::= \langle\text{LowerCase}\rangle | \langle\text{UpperCase}\rangle\]

\[\langle\text{LowerCase}\rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z\]

\[\langle\text{UpperCase}\rangle ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z\]
Extended BNF

Definition

**Extended BNF** or **EBNF** adds to BNF the following conventions:

1. An optional occurrence of some portion of a production rule choice is enclosed in square brackets \([\_]\).
   
   \([u]\) means zero or one occurrence of \(u\) where \(u \in (T \cup N)^+\).

2. An arbitrary number of occurrences of some portion of a production rule choice is enclosed in braces \(\{\_\}\).
   
   \(\{u\}\) means zero or more occurrences of \(u\) where \(u \in (T \cup N)^+\).
Example

In normal BNF, numbers can be defined as follows

<table>
<thead>
<tr>
<th>bnf</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules</td>
<td></td>
</tr>
<tr>
<td>⟨Number⟩</td>
<td>::= ⟨Digit⟩</td>
</tr>
<tr>
<td>⟨Digits⟩</td>
<td>::= ⟨Digit⟩</td>
</tr>
<tr>
<td>⟨Digit⟩</td>
<td>::= 0</td>
</tr>
<tr>
<td>⟨NonZeroDigit⟩</td>
<td>::= 1</td>
</tr>
</tbody>
</table>
### Example

In EBNF, we can define them as follows:

<table>
<thead>
<tr>
<th>ebnf</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules</td>
<td></td>
</tr>
<tr>
<td>⟨Number⟩</td>
<td>::= ⟨Digit⟩</td>
</tr>
<tr>
<td>⟨Digit⟩</td>
<td>::= 0</td>
</tr>
<tr>
<td>⟨NonZeroDigit⟩</td>
<td>::= 1</td>
</tr>
</tbody>
</table>
Lemma

Let $L \subseteq T^*$ be a language. $L$ is definable in EBNF iff it is definable in BNF.

Proof: Exercise.