Introduction to BNF

Cezar Ionescu
cezar.ionescu@th-deg.de

WS 2023-24

Outline

1. Admin
2. Backus-Naur Form

Admin

Questions?

Backus-Naur Form

Programming Language Syntax

(Still taken from clip of John Backus group meeting (1989))

John Backus (1934-2007) was one of the major developers of FORTRAN, the oldest programming language still in use.

The first description of FORTRAN was carried out in everyday English.

Programming Language Syntax

An example from the first FORTRAN report (1954):

A. GENERAL FORM:

Three or more alphabetic or numeric characters (beginning with an alphabetic character) followed by a left parenthesis followed by 1st argument followed by a right parenthesis or by a comma followed by 2nd argument followed by a right parenthesis or by a comma followed by 3rd argument, etc.

B. EXAMPLES:

i) sin(a)

ii) sqrt(a+b) : means $\sqrt{a+b}$

iii) factl(m+n) : means (m+n) !

BNF

- 1959: John Backus introduced a formalism that could be used to define the *syntax* of programming languages in a way that was both *simpler* and *more precise*.
- 1960: Peter Naur (1928-2016) extended Backus' formalism and used it in the „ALGOL 60 Report“
- The resulting formalism is called *Backus-Naur-Form* (BNF).

References:

- Appendix 1 in *Science of Programming*, David Gries, 1981.
- Appendix B.1 in *Artificial Intelligence–A Modern Approach*, Russell and Norvig, 4th Ed., 2020 (but any edition contains this appendix).
- Pages 12-17 from Principles of Programming Languages, lecture notes by Graham Hutton, Univ. of Nottingham.

Example

BNF fragment from the „ALGOL 60 Report“:

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<unsigned integer> ::= <digit> | <unsigned integer><digit>
```

Informally: a digit is either 0, or 1, ..., or 9. An unsigned integer is either a digit, or an unsigned integer followed by a digit.

A similar example in Python.

Terminals, Non-Terminals, Production Rules

- *Non-terminal symbols* such as `<digit>` cannot appear in the "sentences" of the language being described, but are only used to *describe* the sentences.
- *Terminal symbols* such as `8` *can* appear in the sentences of the language being described.
- *Production rules* have a non-terminal on the left side of `::=` and their right-hand side describes possible *re-writings* of that non-terminal.

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<unsigned integer> ::= <digit> | <unsigned integer> <digit>
```

- The above examples shows two production rules, ten terminals, and two non-terminals.

Production Rules

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<unsigned integer> ::= <digit> | <unsigned integer> <digit>
```

- The two rules make up the “grammar” of the “little language” of unsigned integers.
- The *sentences* of this language are strings made up of the terminal symbols `0`, …, `9`.
- The sentences are *derived* from the non-terminal `<unsigned integer>`, by the application of the rules.

Example derivation

```
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    <unsigned integer> ::= <digit> | <unsigned integer> <digit>
```

```
<unsigned integer> => <unsigned integer> <digit>       rule 2
                   => <unsigned integer> 4             rule 1
                   => <unsigned integer> <digit> 4     rule 2
                   => <digit> <digit> 4                rule 2
                   => 1 <digit> 4                      rule 1
                   => 1 7 4                            rule 1
```

This derivation shows that `1 7 4` is a sentence of the language.

Definition of Derivation

Let as, bs, xs be sequences of terminals and non-terminals, <X> ::= xs a rule of the grammar (as, bs, xs may be empty). Then

as <X> bs => as xs bs

denotes a valid *one-step* (or *single*) *derivation*.

The symbol =>* denotes a sequence of *zero or more* one-step derivations. E.g.:

1 <digit> 4 =>* 1 <digit> 4 in zero steps

and

<unsigned integer> =>* 1 7 4 as above

Exercise: “Monkey-Banana Sentences”

```
<Sentence>     ::= <Subject> ' ' <Predicate> ' ' <Object>
<Subject>      ::= <Article_1> ' ' <Adjective_1> ' ' <Noun_1>
<Article_1>    ::= The | A
<Adjective_1>  ::= lazy | young | quick-witted | slow-witted
<Noun_1>       ::= monkey
<Predicate>    ::= eats | wants | finds
<Object>       ::= <Article_2> ' ' <Adjective_2> ' ' <Noun_2>
<Article_2>    ::= a | no
<Adjective_2>  ::= nice | big | ripe | yellow | rotten
<Noun_2>       ::= banana
```

- Identify the terminals and non-terminals of the grammar, and derive five different sentences from the starting symbol.

The example of the monkey-banana sentences shows that “meaningful” sentences can arise from a purely mechanical, syntactical process that does not involve the meanings of the individual words.

Computer Science

The example of the monkey-banana sentences shows that "meaningful" sentences can arise from a purely mechanical, syntactical process that does not involve the meanings of the individual words.

Computer Science (or perhaps better *Comput***ing** *Science*) investigates what can be achieved with purely syntactical means.

Computer Science

The example of the monkey-banana sentences shows that "meaningful" sentences can arise from a purely mechanical, syntactical process that does not involve the meanings of the individual words.

Computer Science (or perhaps better *Comput***ing** *Science*) investigates what can be achieved with purely syntactical means.

Computer Engineering attempts to put into practice the results of computer science.

Simple Arithmetical Expressions

```
<expr> ::= <expr> + <expr>
<expr> ::= <expr> * <expr>
<expr> ::= <expr> - <expr>
<expr> ::= ( <expr> )
<expr> ::= <unsigned integer>
```

Exercise: derive `(1 + 2) * 3`

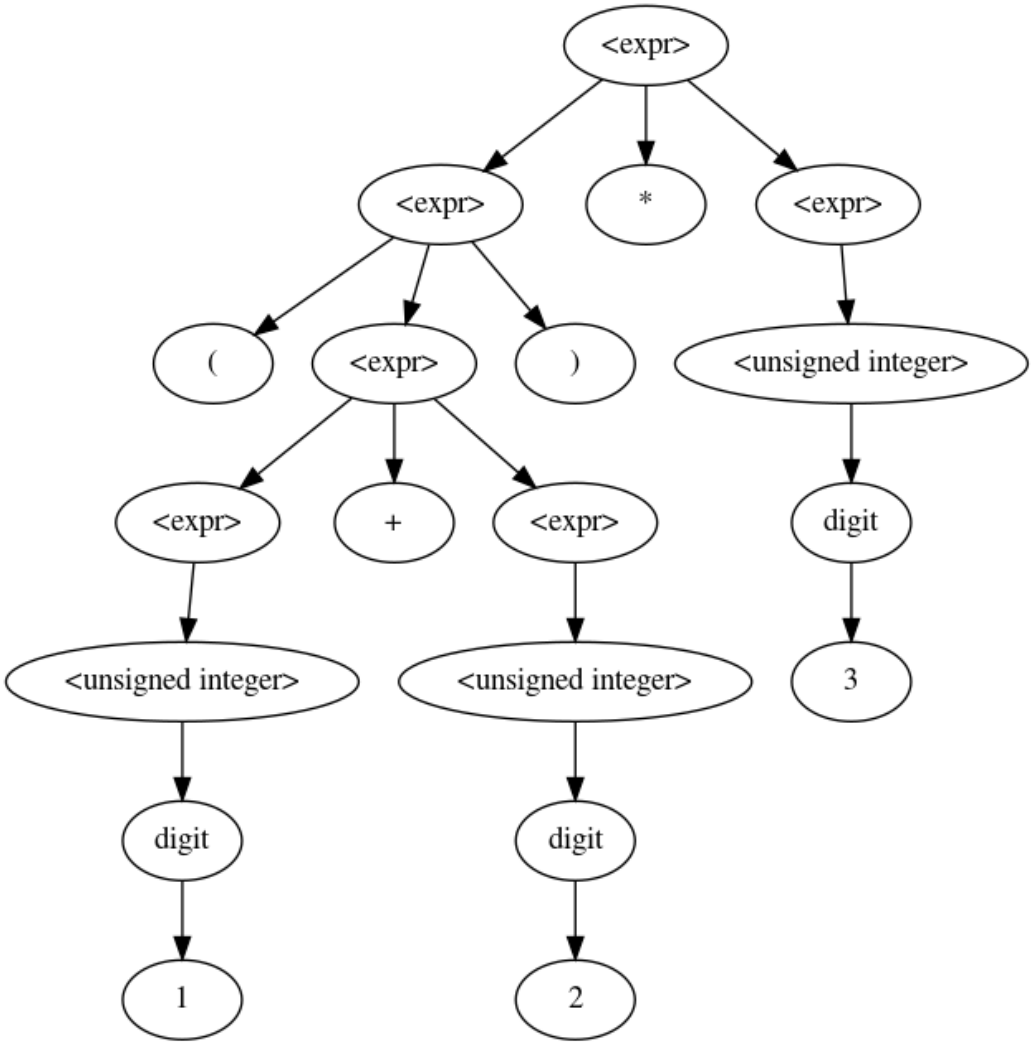
Simple Arithmetical Expressions

One possible solution:

```
<expr> => <expr> * <expr>
       => ( <expr> ) * <expr>
       => ( <expr> + <expr> ) * <expr>
       => ( <unsigned integer> + <expr> ) * <expr>
       => ( <digit> + <expr> ) * <expr>
       => ( <digit> + <unsigned integer> ) * <expr>
       => ( <digit> + <digit> ) * <expr>
       => ( <digit> + <digit> ) * <unsigned integer>
       => ( <digit> + <digit> ) * <digit>
       => ( 1 + <digit> ) * <digit>
       => ( 1 + 2 ) * <digit>
       => ( 1 + 2 ) * 3
```

Syntax trees

Derivation as a syntax tree:



Syntax Trees

A single derivation using the rule <X> ::= xs is represented by a node <X>, with arrows pointing down to the symbols of the sequence xs.

The same syntax tree may represent different derivations. Two derivations that are represented by the same syntax tree are called *equivalent*.

Parsing a sentence means attempting to find a syntax tree corresponding to one of its derivation.

Ambiguous Grammars

A grammar that allows more than one syntax tree (more than one *parse*) for some sentence is called *ambiguous*.

Example: find two different ways to parse

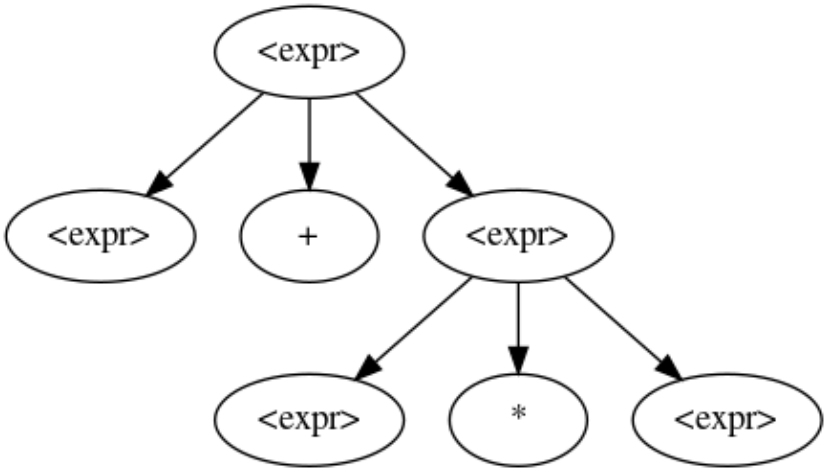
```
<expr> + <expr> * <expr>
```

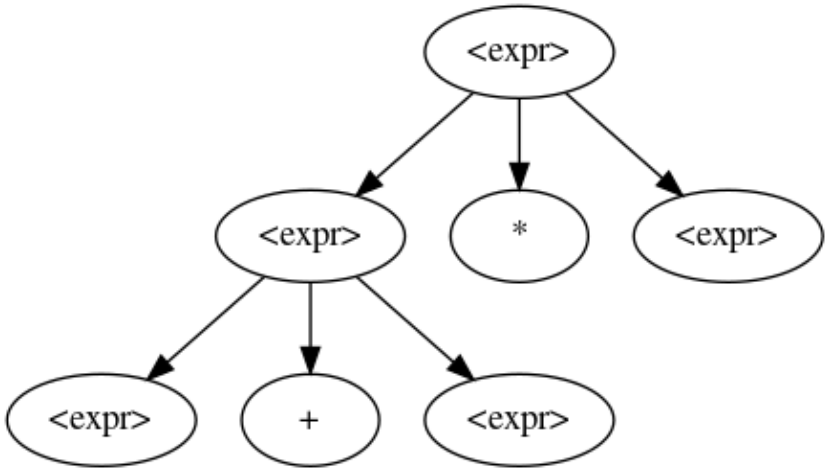
Ambiguous Grammars

A grammar that allows more than one syntax tree (more than one *parse*) for some sentence is called *ambiguous*.

Example: find two different ways to parse

```
<expr> + <expr> * <expr>
```





Eliminating Ambiguity

We can eliminate ambiguity by introducing rules to express *precedence rules*.

```
<expr> ::= <term> | <expr> + <term> | <expr> - <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= <unsigned integer> | ( <expr> )
```

Exercise: A Grammar for Propositional Logic

Develop a grammar for expressions of propositional logic, such as

$$\mathtt{P},\ \mathtt{P} \to \mathtt{Q},\ \mathtt{P} \wedge (\mathtt{Q} \vee \neg\mathtt{P}),\ \mathtt{R} \leftrightarrow \mathtt{Q} \leftrightarrow \neg\mathtt{P}$$

The operator precedence should be (in decreasing order) $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$.

Draw a derivation tree for $\neg(\mathtt{P} \wedge \mathtt{Q}) \leftrightarrow \neg\mathtt{P} \vee \neg\mathtt{Q}$.

Solution

```
<expr>    ::= <expr> ↔ <impl> | <impl>
<impl>    ::= <impl> → <disj> | <disj>
<disj>    ::= <disj> ∨ <conj> | <conj>
<conj>    ::= <conj> ∧ <literal> | <literal>
<literal> ::= <atom> | ¬<literal>
<atom>    ::= <satz> | ( <expr> )
<satz>    ::= P | Q | R | ...
```

Extended BNF

BNF rules can sometimes be awkward:

```
<Zahl> ::= <Positive Zahl> | - <Positive Zahl> | 0
<Positive Zahl> ::= <Ziffer ausser Null><Optionale Ziffernfolge>
<Optionale Ziffernfolge> ::= <Ziffer> <Optionale Ziffernfolge> |
<Ziffer ausser Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer> ::= 0 | <Ziffer ausser Null>
```

(example in Wikipedia (German))

Extended BNF

EBNF extends BNF with:

- Optional expressions : `[ X ]` means that `X` may occur, but it may also be omitted.
- Repetition: `{ X }` means that `X` may occur *zero or more* times.
- Fixed number of repetitions: `3*X` means that `X` must occur exactly three times in succession (XXX).
- Parantheses for grouping : `X | {(Y | Z)}`

EBNF uses `,` to separate the elements of an enumeration. Production rules end with `;`.

For a complete list of extensions (and alternative versions), see thi list of BNF/EBNF variants.

Remark: There are no strict standards for BNF/EBNF (cf. Russell and Norvig).

Extended BNF

BNF:

```
<Zahl> ::= <Positive Zahl> | - <Positive Zahl> | 0
<Positive Zahl> ::= <Ziffer ausser Null><Optionale Ziffernfolge>
<Optionale Ziffernfolge> ::= <Ziffer> <Optionale Ziffernfolge> |
<Ziffer ausser Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer> ::= 0 | <Ziffer ausser Null>
```

EBNF:

```
Zahl = ([ "-" ], ZifferAusserNull, { Ziffer }) | "0" ;
ZifferAusserNull = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;
Ziffer = "0" | ZifferAusserNull ;
```

(example in Wikipedia (German))