Introduction to Programming

Janis Hutz
https://janishutz.com

January 24, 2026

Contents

1 EBNF

= Extended Backus Naur Form / Extended Backus Normal Form

- Can be used to describe automatic tests

1.1 Rules

EBNF-Rule **Definition 1.1**

A rule is described as follows
`LHS <- RHS`

where LHS is the left hand side, which gives the name of the rule. That name is usually written in italic or inside $<...>$. The RHS is the description of the rule using elements, which are literals, names of other EBNF rules defined previously or after or a combination of the four control forms

A character as EBNF rule would be something like

```
<digit_zero> <- 0
```

Another EBNF rule as RHS would be something like:

```
<number_zero> <- <digit_zero>
```

We have to also define an entry rule, which, if nothing else is specified, is the last rule. It marks the entry point from where the checks are run.

1.2 Control forms

1.2.1 Sequence

Sequence **Definition 1.2**

Concatenation of the elements. Noted with a space between the elements.
`<rule> <- E1 E2 E3`

1.2.2 Decision

This control form has two options, namely selection and option.

Selection **Definition 1.3**

We can choose *exactly* one of the elements listed with pipe characters (called stroke).
`<rule> <- E1 | E2 | E3`

Option **Definition 1.4**

We can either choose to use the element or not. Noted with square brackets. ϵ is the empty element, which is returned if the option is not chosen.
`<rule> <- [E]`

1.2.3 Repetition

Repetition **Definition 1.5**

Repeat an element $n \in \mathbb{N}_0$ times, so repeating it 0 times is also valid. Noted using curly brackets.
`<rule> <- {E}`

1.2.4 Recursion

Recursion **Definition 1.6**

Recursively repeat an element, by adding it to the RHS of itself
`<rule> <- <digit> <rule>`

We can have direct or indirect recursion, where indirect recursion is when the recursion is not directly on the rule itself, but through elements of the RHS of the rule.

1.3 Parenthesis

There are precedences, but for where they have not been covered, parenthesis are used to clarify what is meant. Always put them there, if it is not 100% clear what is meant.

Covered precedences: Sequence binds weaker than everything else.

1.4 Special caracters

We use boxes around them if we want to use them as literals, not as EBNF characters, i.e. if we want to write {}, then we write $\boxed{\{\}}$, same goes for space, which we can also write using the space symbol, $\textvisiblespace$

1.5 Nomenclature & Notation

EBNF **Terms**

- ***Legal***: A word is considered legal if there is a derivation of the character sequence
- ***Derivation***: Sequence of derivation steps
- ***Derivation step***:
  - Replace rules with their definition (RHS)
  - *Selection*: Select element in a selection
  - *Option*: Choose whether to select an optional element
  - *Repetition*: Decide on the number of repetitions of the element

1.5.1 Notation of derivation

Derivation **Notation**

- Derivation table
  - First row is the entry rule
  - Last row is a sequence of characters
  - Transition between rows is a derivation step
- Derivation tree
  - Root is the name of the entry rule
  - Leafs are characters
  - Connections are the derivation steps

1.5.2 Syntax vs. Semantics

Syntax = Structure / Form (Grammar), Semantics = Meaning / Interpretation

1.6 Graphical noatation of EBNF

With syntax graph. We can replace names in the syntax graph with another graph.

Sequence: `A B C` $\Longrightarrow$ → A - B - C →

Selection: `A | B | C` $\Longrightarrow$ → Parallel circuit-like notation →

Option: `[A]` $\Longrightarrow$ → Parallel circuit-like notation, two options, top is the element, bottom is empty →

Repetition: {A} $\Longrightarrow$ → Similar to option, but there is a loop back at the top element (allowing repetition) →

2 Peculiarities of Java

- Automatic casting in additions works for all data types (including primitives)
- Automatic casting in comparisons only works for `int` (as well as `short` and `long`, but with `long` we lose precision on conversion) to `double`
- `Array.toString()` outputs the memory address.
- `Class.toString()` is automatically called when outputing to stdout or casting to `string`.
- `Arrays.toString()` outputs the array's elements with `.toString()` (if non-primitive), use `Arrays.deepToString()` to fix
- `File` API is as ugly as a Zombie. Works as follows: `Scanner sc = new Scanner(new File('/path/to/file'));` then using normal Scanner api is possible (e.g. `sc.nextLine()`)
- Accessing `super`'s private fields is compiler error
- Using an implicit constructor of `super` which has a different implementation causes a runtime, not compile time error.
- Abstract class without methods is valid.
- Compile time error if two classes do not share a `super` class when using `instanceof` (because that would evaluate to false at all times)
- `super.x` fine, `super.super.x` not (compiler error)
- `StackOverflow` error can be caught by try-catch
- Method overloading instead of optional parameters
- Incrementation of variable binds stronger than addition. If we have $x = 2$ and calculate `++x + x++`, we get 6, $x = 4$, thus, we have $3 + 3$, as x is first incremented, then added (pre-increment). $++x$ first increments, then the incremented value returned. $x++$ is first returned, then incremented.
- Do-While Loops... I mean wtf (though they can come in handy if you want the loop body to always execute at least once)
- We can change the visibility of a method of a class from not set to *public*
- We can cast a class to a super class if we then only call the methods also defined in said super class. If the method is overridden, the overridden method (the one from the sub-class) is called.
- We can't cast to a subclass (which makes sense), but it's Runtime, not Compile time error (because Java's compiler does type erasing and hence can't check that. Pure idiocy)
- The compiler considers the reference type for an object reference when determining if `instanceof` is possible or not
- Casting a class to a different class of a different branch causes an Exception, not a compile time error
- Casting to an interface or class if there is a connection through a subclass (or subsubclass, ...), runtime rather than compile time error.

The Reference type is the type that the reference has (also called *static type*). Casting a pointer to a different type only changes the reference type, thus all the behvaiour making much more sense:

- Calling methods that are not defined on the reference type → Compile time error

3 Actually resonable stuff in Java

- for, while, try-catch, typing
- Generics (apart from type erasure)
- Automatic type casting exists (even though sorta weird)
- Comparisons between incompatible types is compiler error
- Chaining comparison operators not allowed (but boolean, i.e. `&&` and `||` is allowed)

- Objects are passed by reference, primitives are copied
- Can't call `var.super.method()` (same with `this` for that matter)

4 Tips & Tricks

- **Last four digits of int**: `number % 10000`
- **Decide if number is even**: `number %` $2 = \begin{cases} 0 & \text{if number is even} \\ 1 & \text{else} \end{cases}$
- **Parenthesis**: Always use explicitly
- **Casting**: `(double) 19 / 5` $= 3.8$ instead of 3 (19 is cast to double, then auto-casting to double of 5 $\rightarrow$ division of doubles)
- **Scanner**: `Scanner scanner = new Scanner(System.in);`, then e.g. `int age = scanner.nextInt();`. It is safest to read input as string and then manually cast / convert to required types, as we can handle additional PEBCAK (Problem Exist Between Chair And Keyboard). This is done using `scanner.nextLine()`.
- **Random**: `Random.nextInt(n)` returns a random number in $[1, n]$
- **De Morgan's Rules**: Use them for inverting the expression (but also kinda unnecessary)
- **Off-by-one errors**: Try to run through with small iteration counts and check edge cases
- **CompareTo method**: `o1.compareTo(o2) < 0 // o1 < o2`

5 Abstract class vs Interface

Points	**Abstract class**	**Interface**
Abstract methods	✓	✓
Concrete methods	✓	X
Static methods	✓	✓
Inheritance count	1 only	multiple
Access modifiers	Any	None
Variables	Member allowed	All `public static final`
Instantiation possible	X	X

Table 5.1: Comparison of abstract classes and interfaces

Abstract classes are used when there is a default implementation of a method that is expected not to be overridden and a different class should only inherit this class and not combine, in any other case Interface, since it's more flexible.

6 Maps, Lists & Sets

6.1 Maps

Similar to plain objects in TS, `Map<KeyType, ValueType> map = new HashMap<KeyType, ValueType>();`.

6.2 Lists

Work similarly to Arrays, but are dynamic size (like Vectors in Rust). Usually, we use ArrayList, but Stack and Queue also implement this interface. `List<Type> list = new ArrayList<Type>();`

6.3 Sets

Sets are like mathematical sets, in that they only allow an element to be in it once. See the different implementations below for details on how they can be implemented. `Set<Type> set = new HashSet<Type>();`

6.4 Different implementations

Maps & Sets share similar implementations for the interfaces.

A TreeSet / TreeMap is based on a binary tree and a class `E` has to implement `Comparable<E>` (i.e. the `compare` method). **Time complexity** $\mathcal{O}(\log(n))$

A HashSet / HashMap uses a hash table (a table that stores a reference to an object of which a hash was computed). The order of elements is not guaranteed and not predictable from system to system **Time complexity** $\mathcal{O}(1)$.

A LinkedHashSet / LinkedHashMap work similarly to HashSet / HashMap, but the order is maintained (i.e. output in the order the elements were added). **Time complexity** $\mathcal{O}(1)$

7 Loop-Invariant

1. Write down a table with the loop iteration number and what the state of each variable in the loop is
2. Check what causes the loop to end (in `while`, it's the inverse of the condtion, in `for`, it is the same concept)
3. Then, use that break condition (inverted) and establish an upper (or lower) bound for the variable involved. (e.g. `i < arr.length` in the loop condition turns into `i < 0 arr.length` in the loop invariant)
4. Specify all condtions for all the variables known through the pre and post-conditions, as far as applicable

Also, all other variables that are being changed need to be addressed in the invariant, ensuring that the statements resolve to boolean when executed (`==` and not `=` for comparison (because valid Java Syntax required).

For the pre- and postconditions, ensure to also address ALL variables in the loop, even if it seems unnecessary.

8 Mean things

8.1 In EBNF

- Unbalanced brackets

8.2 Java

- Non-private attributes that could be changed prior to execution
- Spaces in variable names
- Using reserved names as variable names
- Unbalanced brackets