ANSWERS

# A6.5 Backus-Naur Form and syntax diagrams

Theory of computation · A level · AQA 7517 4.4.3.1, Eduqas A500QS 1.8 · about 25 min

## What this lesson is about

Production rules, syntax diagrams, a grammar for robot programs, recursive descent, and why BNF can describe what a regex cannot.

- Production rules
- A robot command language
- Syntax diagrams
- Why BNF can do what a regex cannot
- Checking syntax with recursion

## Questions

**6 marks in all**

**1.** In BNF, what does ::= mean? [1 mark]

- ☐ **A** Is equal to
- ☐ **B** Is defined as
- ☐ **C** Or
- ☐ **D** Is followed by

> **Answer:** B. A production rule defines the non-terminal on the left as the alternatives on the right, separated by |.

**2.** Using <integer> ::= <digit> | <digit><integer> and <move> ::= <direction><integer> with <direction> ::= F | B | L | R, which are valid moves? [1 mark]

Tick every answer that is true.

- ☐ **A** F20
- ☐ **B** R7
- ☐ **C** F
- ☐ **D** 20F
- ☐ **E** L005

> **Answer:** A, B, E. A move is one direction letter then one or more digits. L005 is fine: the rule does not forbid leading zeros.

**3.** Why can BNF describe some languages that regular expressions cannot? [1 mark]

- ☐ **A** BNF allows alternatives with |
- ☐ **B** BNF rules can be recursive in the middle of a rule, so they can describe nesting such as balanced brackets
- ☐ **C** BNF uses angle brackets
- ☐ **D** BNF can only describe finite languages

> **Answer:** B. A rule like <s> ::= ab | a<s>b wraps matching symbols round the middle. A regex has repetition but cannot match counts.

**4.** In a syntax diagram, what does a path that loops back round a box show? [1 mark]

- ☐ **A** The item may be skipped
- ☐ **B** The item may be repeated
- ☐ **C** The item is a terminal
- ☐ **D** The syntax is invalid

> **Answer:** B. Following the loop lets you pass through the box again as many times as you like.

**5.** Given <s> ::= ab | a<s>b, how many times must the rule be used to produce aaabbb? [1 mark]

> **Answer:** 3. a<s>b twice wraps two pairs round ab: a(a(ab)b)b. That is three uses of the rule.

**6.** What does this program print? [1 mark]

```
def s(text, i):
    if text[i:i + 2] == "ab":
        return i + 2
    if text[i:i + 1] == "a":
        j = s(text, i + 1)
        if j != -1 and text[j:j + 1] == "b":
            return j + 1
    return -1

for t in ["aabb", "aab", "abb"]:
    print(t, s(t, 0))
```

> **Answer:**
>
> ```
> aabb 4
> aab -1
> abb 2
> ```
>
> aabb matches all 4 characters. aab fails with -1. abb matches only ab, returning 2, so the whole string is not valid.

## The task: check robot programs

Write a recursive descent checker for the robot command language above and use it on the strings in `tests`. - Write one function per rule you need, each taking `(text, i)` and returning the position after the match, or -1 if it does not match. At least `program`, `command` and `integer`. - A string is valid only if `program` matches all of it. - Check the nesting with your functions. Do not use the `re` module: no regex can check matched brackets. - For each string in `tests`, in order, print the string, a space, then `valid` or `invalid`. For example `F20R90 valid`. Eight lines in all.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

tests = ["F20R90", "[4F20R90]", "[2F10[3L5]]", "F20R", "[4F20", "4F20", "[3]", "F1]"]

def integer(text, i):
    return -1
```

**The hint students can ask for:** Follow the BNF: each function tries its rule's alternatives at position i. A command is a move if the character there is a direction letter, or a repeat if it is an opening bracket. After one command, a program carries on only if the next character could start another command. A repeat must end with its own closing bracket.

### A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

tests = ["F20R90", "[4F20R90]", "[2F10[3L5]]", "F20R", "[4F20", "4F20", "[3]", "F1]"]

def integer(text, i):
    # <integer> ::= <digit> | <digit><integer>
    if text[i:i + 1].isdigit():
        j = integer(text, i + 1)
        return j if j != -1 else i + 1
    return -1

def command(text, i):
    # <command> ::= <move> | <repeat>
    ch = text[i:i + 1]
    if ch != "" and ch in "FBLR":
        return integer(text, i + 1)
    if ch == "[":
        j = integer(text, i + 1)
        if j == -1:
            return -1
        k = program(text, j)
        if k != -1 and text[k:k + 1] == "]":
            return k + 1
    return -1

def program(text, i):
    # <program> ::= <command> | <command><program>
    j = command(text, i)
    if j == -1:
        return -1
    nxt = text[j:j + 1]
```

```
    if nxt != "" and nxt in "FBLR[":
        return program(text, j)
    return j

for text in tests:
    print(text, "valid" if program(text, 0) == len(text) else "invalid")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.