Group Assignment 1 - The Vault Runner

AI1030 - Python Programming

1 Scenario: "The Vault Runner"

You must design a programming language to instruct a small robot (*Runner*) navigating a vault-like 2D grid world. The Runner must collect keys and escape through a door, using only the instructions provided by your language. **There is only one room for the Runner to navigate. The exit tile serves as a direct shortcut out of the vault: if the Runner reaches it, the escape is complete without needing to collect the key or open the door.** Otherwise, the Runner must first collect the key, use it to open the door, and then continue to the exit.
The Vault Runner's world is a 2D grid with walls, empty floor, a key (K), a door (D), and an exit (E). The robot can sense only its immediate surroundings (e.g., whether the front is clear) and must use simple instructions to navigate and escape. This environment is designed to teach programming language design under constraints.

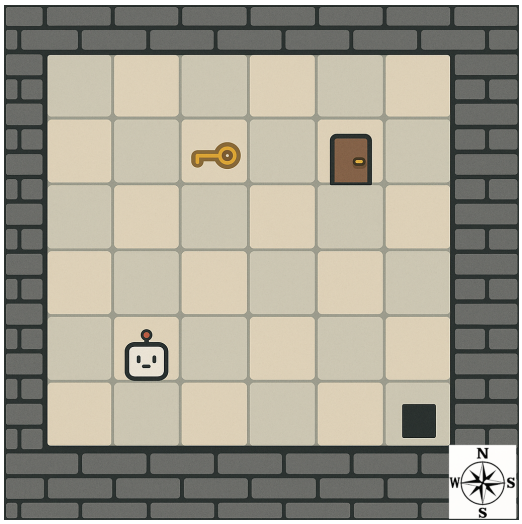

Figure 1: Example of 2D environment with walls, a key, a door, an exit, and the robot.

World Basics (fixed)

- Tiles: `floor`, `wall`, `key`, `door`, `exit`.
- Sensors: `front_is_clear`, `on_key`, `at_door`, `at_exit`.
- Actions: `move_forward`, `turn_left`, `turn_right`, `pick_key`, `open_door`. In addition, you may utilize a **random turn** action, where the robot randomly turns either left or right.
- Room/Corridor: There is only one rectangular room surrounded by walls. There are no other obstacles. For the first program that you should deliver, the world is a corridor of one tile width, contains turns, and is blocked at both ends (see Section 2B).

Constraints (non-negotiable)

1. Stick to the sensor outputs as variables; only literals, **a single accumulator/flag**.
2. Language alphabet: ≤ 20 distinct keywords/symbols.

2 Tasks

Part A: Language Design (Specification)

Produce a concise specification (max 3 pages):

- Purpose & design principles.
- Token set and grammar.
- Semantics of constructs and sensors.
- **Extension (optional)**. Support for memory/state and also support for variables, flags, any additional construct you would like to include in your design.

Part B: Pre-set Programs

Write two programs (plus an optional one) in your language, including code, token count, and short explanations. The programs are as follows.

1. Navigate a **twisting corridor** (that is, a corridor with **turns**). The corridor is blocked at both ends. Inside the corridor there is a key, a door, and an exit hatch. The runner can escape using either the exit hatch or the door (the key is required to open the door). The initial position of the robot is **unknown**, as well as the direction of facing.
2. Find and collect the key, open the door and escape, or escape from the exit hatch (whatever comes first). The world is an orthogonal rectangular room with four surrounding walls. There are no obstacles in the room. Initially, the robot is located on the lower-left tile of the room **facing north**.
3. **Extension (optional)**. Solve a map where there are multiple keys, one door, and one exit hatch. Only one of the keys opens the door. The world is an orthogonal rectangular room with four surrounding walls. There are no obstacles in the room. The initial location of the robot is **unknown** (could be on any tile), as well as the facing direction.

Part C: Reference Implementation

Implement a small interpreter for your language in the Python programming language (see also the supplement). Requirements:

- Tokenizer, parser (token limits enforced). Basically, the tokenizer should be able to receive commands in your language and identify the corresponding tokens. In addition, grammatical rules must be enforced.
- Executor mapped to API. In its simplest form, the output of the interpreted program could be a set of **print() commands** stating which action is taken by the robot (see supplement).
- Input. For simplicity, you can assume that the input to the interpreter is given as a **list of strings** at the beginning of the Interpreter code. Then, the Interpreter should scan the list, element-wise, and proceed with tokenization and parsing. In case of a syntax error, a relevant message must be printed.

Part D: Reflection

Write 300–500 words on:

- Trade-offs you made.
- How constraints shaped the design.

3 Deliverables

Each group submission must include the following parts:

3.1 Code (40% of grade)

- A working interpreter in Python.
- At least two pre-set programs (look at PartB) in your language.
- A small test suite (3 cases).
- The code should be well-structured, commented, and enforce all constraints.

3.2 Report (40% of grade)

- **Language Specification** (purpose, tokens, grammar, semantics, error model).
- **Examples** (with token counts and explanations).
- **Reflection** (300–500 words on design trade-offs and how constraints shaped the language).

3.3 Recorded Group Presentation (20% of grade)

- Duration: 8–10 minutes.
- Every member of the group must appear and contribute.
- Present:
  - Key features of your language.
  - An example program.
  - A short demo of your interpreter.
  - Reflections on what you learned about language design.
- Format: Pre-recorded video (MP4 file).

4 Grading Rubric

Component	Criteria	Points
Code (40%)	- Interpreter correctness and functionality - At least 2 working example programs - Enforces constraints - Includes 3 test cases - Code readability and comments	40
Report (40%)	- Clear language specification (tokens, grammar, semantics) - Examples with explanations and token counts - Reflection (300–500 words) on trade-offs and constraints - Professional structure and clarity	40
Presentation (20%)	- 8–10 minutes, well-structured - All group members contribute - Demonstrates key language features and a sample program - Interpreter demo included - Clear articulation of lessons learned	20
Extension (5%)	- Extended language using more features as shown in Section 2, Part A. - Extended implementation of the corresponding Interpreter including three different test cases.	5
Total		**105**

5 Grade Bands

Grade	GPA	Score Range (%)	Performance Description
A+	4.0	95–100	Exceptional work, complete and polished in all aspects; interpreter, report, and presentation exceed expectations.
A	4.0	90–94	Excellent work, very strong across all components with only minor improvements possible.
A-	3.7	85–89	Very good work, complete with small gaps in detail, clarity, or testing.
B+	3.3	80–84	Good work, most requirements met with some issues in implementation or explanation.
B	3.0	75–79	Satisfactory work, functional but with gaps in coverage or clarity.
B-	2.7	70–74	Adequate work, significant issues in one component (e.g., interpreter or report).
C+	2.3	65–69	Passable work, limited testing or reflection, some incomplete elements.
C	2.0	60–64	Barely sufficient, multiple weaknesses across code, report, or presentation.
C-	1.7	55–59	Weak performance, incomplete work with major flaws but some evidence of effort.
D+	1.3	50–54	Very limited work, serious issues across most components, minimal demonstration of understanding.
D	1.0	40–49	Poor work, interpreter largely non-functional or report missing key sections.
F	0.0	0–39	Unsatisfactory, major requirements missing or not attempted.