Exercise walk-through

20.1.1

The five addressing modes, and declarative programming with facts, rules and goals

A-Level Computer Science

You must be able to

- write and trace low-level code that uses the **immediate**, **direct**, **indirect**, **indexed** and **relative** addressing modes
- solve a problem by writing appropriate **facts 事实** and **rules 规则** from supplied information
- write code that can satisfy a **goal 目标** using facts and rules
- identify the paradigm a code sample belongs to, and describe the characteristics of each

1 The method

Method — The five addressing modes

Every one of these instructions writes the **same operand**, 105. What differs is how many times the processor follows it.
Take this memory, with the index register `IX` holding 2:

Address	**27**	**105**	**106**	**107**
Contents	91	27	64	43

Instruction	**Mode**	**What `ACC` becomes**	**Why**
`LDM #105`	**immediate**	**105**	the operand **is** the value; # marks it
`LDD 105`	**direct**	**27**	the operand is the **address** of the value
`LDI 105`	**indirect**	**91**	105 holds 27, and 27 is the address actually used
`LDX 105`	**indexed**	**43**	the address is $105 + \texttt{IX} = 107$
`JMR +6`	**relative**	–	jump to an **offset** from the current instruction, not to a fixed address

Method — The five addressing modes

- Count the steps: immediate follows the operand **zero** times, direct **once**, indirect **twice**. That is the whole distinction, and it is what a trace question tests.
- **Indexed** exists for **arrays**: put the array's start address in the instruction and the subscript in `IX`, then increment `IX` to step along it, with the instruction itself unchanged.
- **Relative** exists for **relocatable** code: the program works wherever it is loaded, because every jump is measured from where it already is rather than from address zero.

Method — The five addressing modes

Every one of these instructions writes the same operand , 105.

address	contents
105	27
106	64
107	105
27	91
145	16

mode	instruction	ACC receives
immediate	`LDM #105`	105 (the operand itself)
direct	`LDD 105`	27 (contents of 105)
indirect	`LDI 105`	91 (105 holds 27, so contents of 27)
indexed	`LDX 105`	105 (contents of $105 + \mathrm{IX} = 107$)
relative	`JMR +65`	next address $40 + 65 = 105$

$\mathrm{IX} = 2$, instruction at address 40

Method — Declarative programming: facts, rules and goals

In a **declarative** language the program states **what is known** and **what is wanted**. The programmer writes no sequence of steps; the language's **inference engine 推理引擎** searches for a way to satisfy the goal.

- A **fact** states something that is true, with no condition: `book(dracula, stoker).`
- A **rule** states something that is true **provided** other things are: `classic(X) IF published(X, Y) AND Y < 1900.`
- A **goal** is the question asked of the program: `classic(dracula).`

Worked example. *A library holds these facts.*

```
book(dracula, stoker).
book(frankenstein, shelley).
published(dracula, 1897).
published(frankenstein, 1818).
```

Method — Declarative programming: facts, rules and goals

```
classic(X) IF published(X, Y) AND Y < 1900
```

```
classic(W)
```

Method — Declarative programming: facts, rules and goals

To satisfy `classic(W)`, the engine takes each fact matching `published(W, Y)` in turn, binds `Y`, and tests `Y < 1900`. Both succeed, so it returns `dracula` and `frankenstein`.

- A **variable** starts with a capital letter and a **constant** with a small one. Writing `classic(Dracula)` asks a different question – it is a variable, so it matches anything.
- The order of the facts does not change the answer. That is the point of the paradigm: there is no flow of control to get wrong.
- A goal with a **constant** asks "is this true?" and answers yes or no; a goal with a **variable** asks "which values make this true?" and returns them.

Method — Naming the paradigm from a sample

Sample	Paradigm	The giveaway
`LDD 200`, `ADD #5`, `STO 201`	**low-level**	mnemonics, registers, memory addresses
`FOR Count <- 1 TO 10 ... NEXT Count`	**imperative**	a sequence of statements that change state
`CLASS Dog ... PRIVATE Name : STRING`	**object-oriented**	data and methods bound together in a class
`type(lion, wild).` and `dangerous(X) IF type(X, wild)`	**declarative**	facts and rules, with no order of execution

2 Practice

Practice 2.1 ▪ 4 marks

Give the contents of `ACC` after each instruction is executed on its own.

Practice 2.1 (a)

```
LDM #105
```

Solution 2.1 (a)

Method: The five addressing modes — p.4

Worked steps

`LDM #105`: the operand is the value, so `ACC` $= \mathbf{105}$ **B1**

Practice 2.1 (b)

```
LDD 105
```

Solution 2.1 (b)

Worked steps

`LDD 105`: the contents of address 105, so `ACC` $= \mathbf{27}$ B1

Practice 2.1 (c)

```
LDI 105
```

Solution 2.1 (c)

Worked steps

`LDI 105`: 105 holds 27, so the contents of address 27, $\texttt{ACC} = \mathbf{91}$ B1

Practice 2.1 (d)

```
LDX 105
```

Solution 2.1 (d)

Worked steps

`LDX 105`: the address is $105 + 2 = 107$, so `ACC` $= \mathbf{43}$ **B1**

Practice 2.2 ▪ 2 marks

Name the addressing mode used by each instruction in 2.1.

Solution 2.2

Method: The five addressing modes — p.4

Worked steps

- (a) **immediate**, (b) **direct** B1
- (c) **indirect**, (d) **indexed** B1

Practice 2.3 ▪ 2 marks

State the difference between a **fact** and a **rule** in a declarative language.

Solution 2.3

Method: Declarative programming: facts, rules and goals — p.7

Worked steps

- a **fact** states something that is unconditionally true **B1**
- a **rule** states something that is true **only if** one or more other conditions are satisfied **B1**

Practice 2.4 ▪ 3 marks

Name the paradigm each sample belongs to.

Practice 2.4 (a)

`LDD 200` then `ADD #5` then `STO 201`

Solution 2.4 (a)

Method: Naming the paradigm from a sample — p.10

Worked steps

low-level B1

Practice 2.4 (b)

```
CLASS Dog with PRIVATE Name : STRING
```

Solution 2.4 (b)

Worked steps

object-oriented B1

Practice 2.4 (c)

`type(lion, wild).` and `dangerous(X) IF type(X, wild)`

Solution 2.4 (c)

Worked steps

declarative B1

3 Exam-style

Exam-style 3.1 ▪ 4 marks

Explain why **indexed** addressing is used to process an array, and why **relative** addressing makes a program **relocatable**.

Solution 3.1

Method: The five addressing modes — p.4

Worked steps

- in **indexed** addressing the effective address is the operand plus the index register, so the instruction holds the array's **start** address and `IX` holds the subscript **B1**
- incrementing `IX` moves to the next element with the **instruction itself unchanged**, so one instruction inside a loop can read the whole array **B1**
- in **relative** addressing the target is an **offset from the current instruction** rather than an absolute address **B1**
- so wherever the program is loaded in memory, every jump still lands in the right place and nothing has to be recalculated: the code is relocatable **B1**

Exam-style 3.2 ▪ 3 marks

State what is meant by **indirect** addressing, and give **one** situation in which it is needed.

Solution 3.2

Method: The five addressing modes — p.4

Worked steps

- the operand is the **address of an address**: the processor reads the contents of the given address and uses **that** as the address of the value it wants B1
- so it follows the operand **twice** rather than once B1
- it is needed for **pointers**, and wherever the location of the data is not known until run time – for example following a linked list, or reaching a value through a table of addresses B1

Exam-style 3.3 ▪ 8 marks

A vet's surgery records its patients. Each animal has an owner and a species.

Exam-style 3.3 (a) ▪ 4 marks

Write **facts** to record that Bella is a dog owned by Chen, and that Milo is a cat owned by Rani.

Solution 3.3 (a)

Method: Declarative programming: facts, rules and goals — p.7

Worked steps

`animal(bella, dog).` and `owns(chen, bella).` B1 B1

- `animal(milo, cat).` and `owns(rani, milo).` B1 B1
- names are constants and so start with a **small** letter; any consistent predicate names are accepted, provided the same ones are used in (b)

Exam-style 3.3 (b) ▪ 2 marks

Write a **rule** that defines a `dogowner` as someone who owns an animal whose species is dog.

Solution 3.3 (b)

Worked steps

`dogowner(X) IF owns(X, Y) AND animal(Y, dog)` **B1** **B1**

- one mark for the two conditions, one for joining them with the shared variable `Y`, which is what links the owner to their animal

Exam-style 3.3 (c) ▪ 2 marks

Write the **goal** that would list every dog owner, and state what the inference engine returns.

Solution 3.3 (c)

Worked steps

`dogowner(W)` **B1**

- the engine returns `chen`, the only owner whose animal has species dog **B1**

Exam-style 3.4 ▪ 3 marks

Explain how a declarative program differs from an imperative one, in terms of what the programmer writes.

Solution 3.4

Method: Declarative programming: facts, rules and goals — p.7

Worked steps

- in an **imperative** program the programmer writes a **sequence of instructions** that are executed in order and change the program's state: they say **how** the task is done **B1**
- in a **declarative** program the programmer writes **facts and rules** – what is known and what is wanted – and no order of execution at all **B1**
- the language's **inference engine** then works out how to satisfy the goal, so the programmer says **what** is required rather than how to get it **B1**