ANSWERS

A10.8 Stages of compilation, linkers and loaders

Operating systems, software and translators · A level · OCR H446 1.2.2, AQA 7517 4.6.1.3, Eduqas A500QS 1.8 · about 45 min

What this lesson is about

Lexical analysis, syntax analysis, code generation and optimisation, then static and dynamic linking and the loader.

- 1. Lexical analysis
- 2. Syntax analysis
- 3. Code generation
- 4. Optimisation
- Libraries, linkers and loaders

Questions

6 marks in all

1. Put the stages of compilation in order. [1 mark]

Number the lines 1 to 4 to put them in the right order.

___ `Optimisation`
___ `Lexical analysis`
___ `Syntax analysis`
___ `Code generation`

Answer:

```
Lexical analysis
Syntax analysis
Code generation
Optimisation
```

Characters become tokens, tokens become a syntax tree, the tree becomes object code, and the object code is improved.

2. Which of these happen during lexical analysis? [1 mark]

Tick every answer that is true.

- ☐ **A** Comments and whitespace are removed
- ☐ **B** The source is split into tokens
- ☐ **C** Identifiers are added to the symbol table
- ☐ **D** The abstract syntax tree is built

Answer: A, B, C. The syntax tree is built by the syntax analyser, from the tokens.

3. At which stage is a missing closing bracket detected? [1 mark]

- ☐ **A** Syntax analysis
- ☐ **B** Lexical analysis
- ☐ **C** Code generation
- ☐ **D** Linking

> **Answer:** A. Each token is valid on its own; it is their arrangement that breaks the grammar.

4. What is an advantage of dynamic linking over static linking? [1 mark]

- ☐ **A** Executables are smaller, and many programs can share one copy of a library in memory
- ☐ **B** The program will run even if the library is missing
- ☐ **C** The program always uses the exact library version it was built with
- ☐ **D** No loader is needed

> **Answer:** A. The other options describe static linking, or are false.

5. Which part of the operating system copies an executable from secondary storage into memory and adjusts its addresses so it can run? [1 mark]

> **Answer:** loader. The linker builds the executable; the loader puts it in memory and starts it.

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

```
source = "total = total + 5  # add five"
code = source.split("#")[0]
tokens = code.split()
print(len(tokens), tokens)
```

> **Answer:**
>
> ```
> 5 ['total', '=', 'total', '+', '5']
> ```
>
> The comment is thrown away before the rest is split into tokens, as a lexer does.

The task: a lexer

Write `tokenise(text)`, the lexical analysis stage for a tiny language. `text` is a string that may contain several lines. It returns a list of `(type, text)` tuples in order, where `type` is one of: - `"KEYWORD"`: a word in `KEYWORDS`; - `"IDENTIFIER"`: any other word, made of letters, digits and underscores and not starting with a digit; - `"NUMBER"`: one or more digits; - `"OPERATOR"`: one character from `OPERATORS`; - `"PUNCTUATION"`: one character from `PUNCTUATION`. Whitespace is skipped, and a `#` starts a comment that runs to the end of its line and produces no tokens. Do not use Python's own `tokenize` module or regular expressions: read the characters yourself. Then print each token of `source` on its own line as `<type> <text>`, for example `KEYWORD while`. Finally print the symbol table: `symbols:` followed by each different identifier, in the order it first appears, separated by `, `. The robot stays still.

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

KEYWORDS = ["while", "if", "else", "def", "return"]
OPERATORS = "=+-*/<>"
PUNCTUATION = "():,"

source = "while gap > 20:  # keep going\n    gap = gap - step"

def tokenise(text):
    tokens = []
    return tokens
```

The hint students can ask for: Look at one character at a time and let it decide what kind of token is starting. Digits and letters can go on for several characters, so keep reading while the next character still belongs to the same token. A comment runs to the end of its line and produces no tokens.

A solution

```
from bugbot import *
connect()

KEYWORDS = ["while", "if", "else", "def", "return"]
OPERATORS = "=+-*/<>"
PUNCTUATION = "():,"

source = "while gap > 20:  # keep going\n    gap = gap - step"

def tokenise(text):
    tokens = []
    i = 0
    while i < len(text):
        ch = text[i]
        if ch == "#":
            while i < len(text) and text[i] != "\n":
                i = i + 1
        elif ch.isspace():
            i = i + 1
        elif ch.isdigit():
            start = i
            while i < len(text) and text[i].isdigit():
                i = i + 1
            tokens.append(("NUMBER", text[start:i]))
```

```
        elif ch.isalpha() or ch == "_":
            start = i
            while i < len(text) and (text[i].isalnum() or text[i] == "_"):
                i = i + 1
            word = text[start:i]
            tokens.append(("KEYWORD" if word in KEYWORDS else "IDENTIFIER", word))
        elif ch in OPERATORS:
            tokens.append(("OPERATOR", ch))
            i = i + 1
        elif ch in PUNCTUATION:
            tokens.append(("PUNCTUATION", ch))
            i = i + 1
        else:
            raise ValueError("unexpected character " + ch)
    return tokens

symbols = []
for kind, text in tokenise(source):
    print(kind, text)
    if kind == "IDENTIFIER" and text not in symbols:
        symbols.append(text)
print("symbols:", ", ".join(symbols))
```

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