Compiler

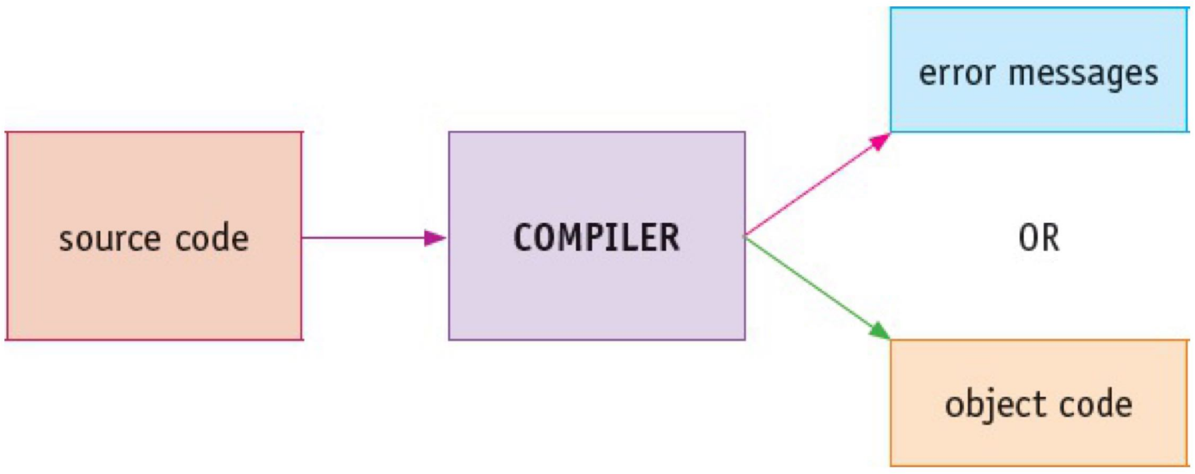


Figure 16.19

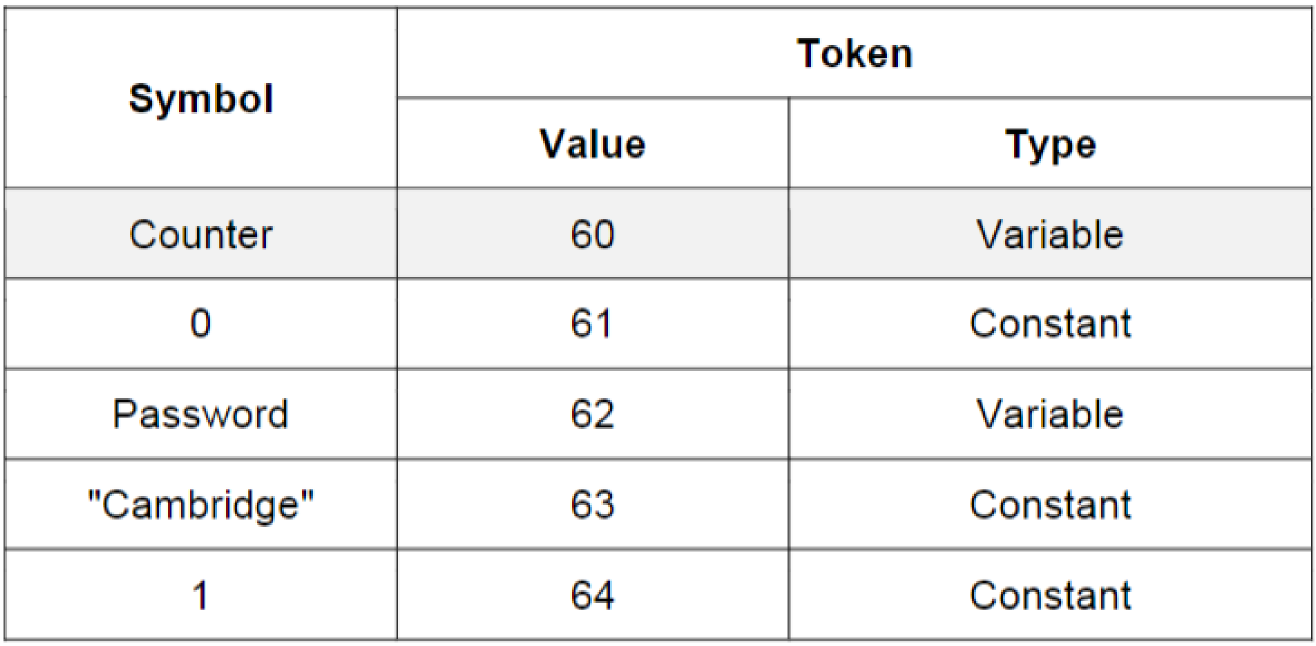
Compilation stages

Lexical analysis

- All necessary characters not required by the compiler, such as the white space and comments, are removed
- The white space removed // redundant characters are removed // removal of comments // identification of errors
- Tokenization
  - Using a keyword table that contains all the tokens for reserved keywords and symbols
  - Convert the source program into tokens
  - Keyword table
    - The reserved keyword used
    - The operator used

Keyword	Token
←	01
+	02
=	03
<>	04
IF	4A
THEN	4B
ENDIF	4C
ELSE	4D
REPEAT	4E
UNTIL	4F
TO	50
INPUT	51
OUTPUT	52
ENDFOR	53

        - Their matching token
- Variables, constants, and identifiers added to a symbol table, and are then converted into locations/address
    - Symbol table
        - Identifier name used
        - The data type
        - Role: e.g. Constant, array, procedure
        - Location marker, value of constant



Symbol	Token	
	Value	Type
Counter	60	Variable
0	61	Constant
Password	62	Variable
"Cambridge"	63	Constant
1	64	Constant

Explain how the keyword table and symbol table are used to translate the source code program

- Keywords are looked up in the keyword table
- They are represented by tokens
- Identifiers are looked up in the symbol table
- They are converted into location/address
- Used to create a sequence of tokens (for the program)

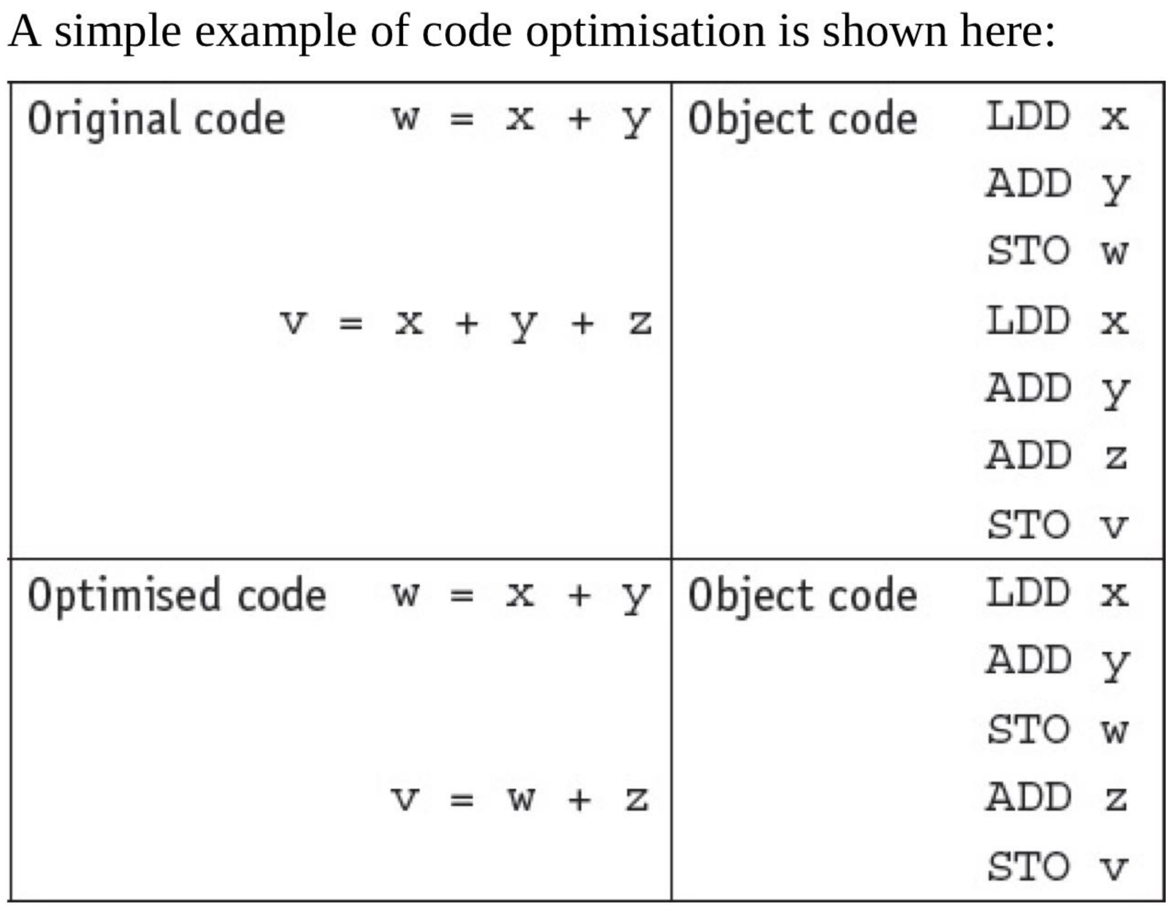
Syntax analysis

- Output from the lexical analysis is checked for grammatical/syntax errors - `parsing`
- The rules for `parsing` can be set out in Backus-Naur form(BNF) notation
- If errors are found: each statement and the associated error is outputted, but the next stage, code generation, will not be attempted
- If no error is found: passed to the next stage of compilation
- `--`
- Construction of a parse tree / parsing
- Checking that the rules of grammar/syntax have been obeyed
- Production of an error report

Code generation

- Produces an object program to perform the task defined in the source code
- The object program is in machine-readable form(binary):
    - Either in machine code that can be directly executed by the CPU
    - Or in intermediate code that is converted into machine code when the program is loaded

Optimization

- Performing the task using the minimum amount of resources
  - Execution time, storage space, memory, and CPU use.
- 

A simple example of code optimisation is shown here:

Original code	`w = x + y`	Object code	`LDD x`
			`ADD y`
			`STO w`
	`v = x + y + z`		`LDD x`
			`ADD y`
			`ADD z`
			`STO v`
Optimised code	`w = x + y`	Object code	`LDD x`
			`ADD y`
			`STO w`
	`v = w + z`		`ADD z`
			`STO v`

Why code is optimized:

- Redundant code removed
- Program requires less memory
- Code reorganized to make it more efficient
- Program will complete task in a shorter time

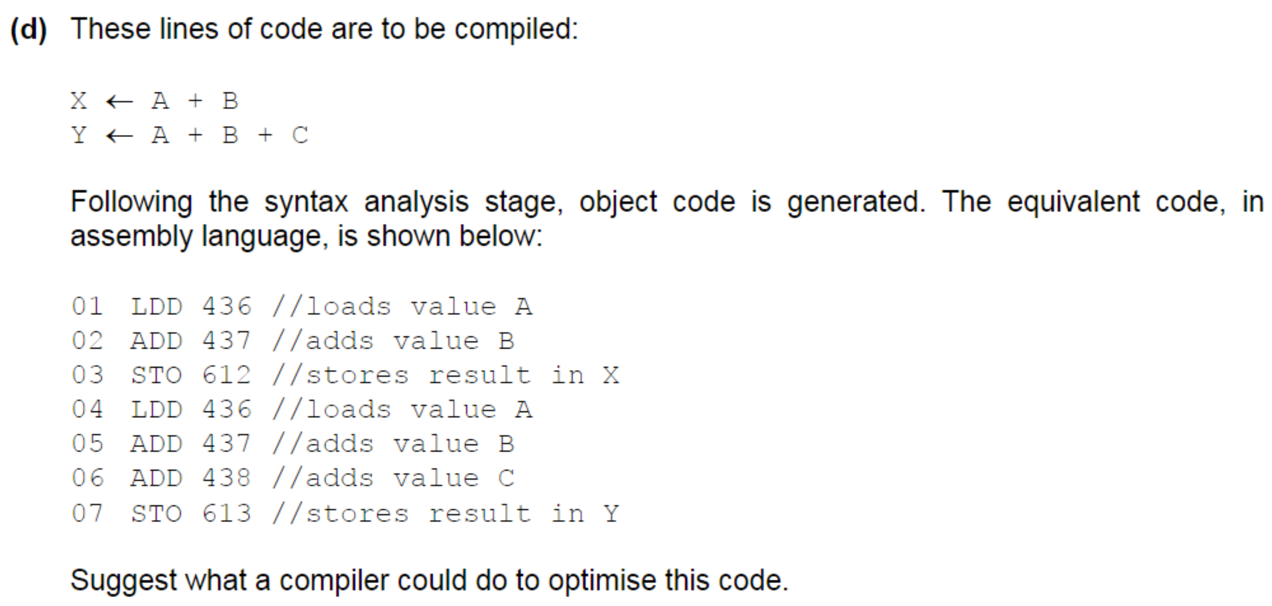
Why optimization is necessary

- Optimisation means the code would have fewer instructions
- Optimised code occupies less memory in space
- Fewer instructions reduce the execution time of the program

Benefits

- Code has fewer instructions/occupies less memory in space
- Shortens the execution time for the program // time taken to execute whole program decreases

Past-paper questions



(d) These lines of code are to be compiled:

```
X ← A + B
Y ← A + B + C
```

Following the syntax analysis stage, object code is generated. The equivalent code, in assembly language, is shown below:

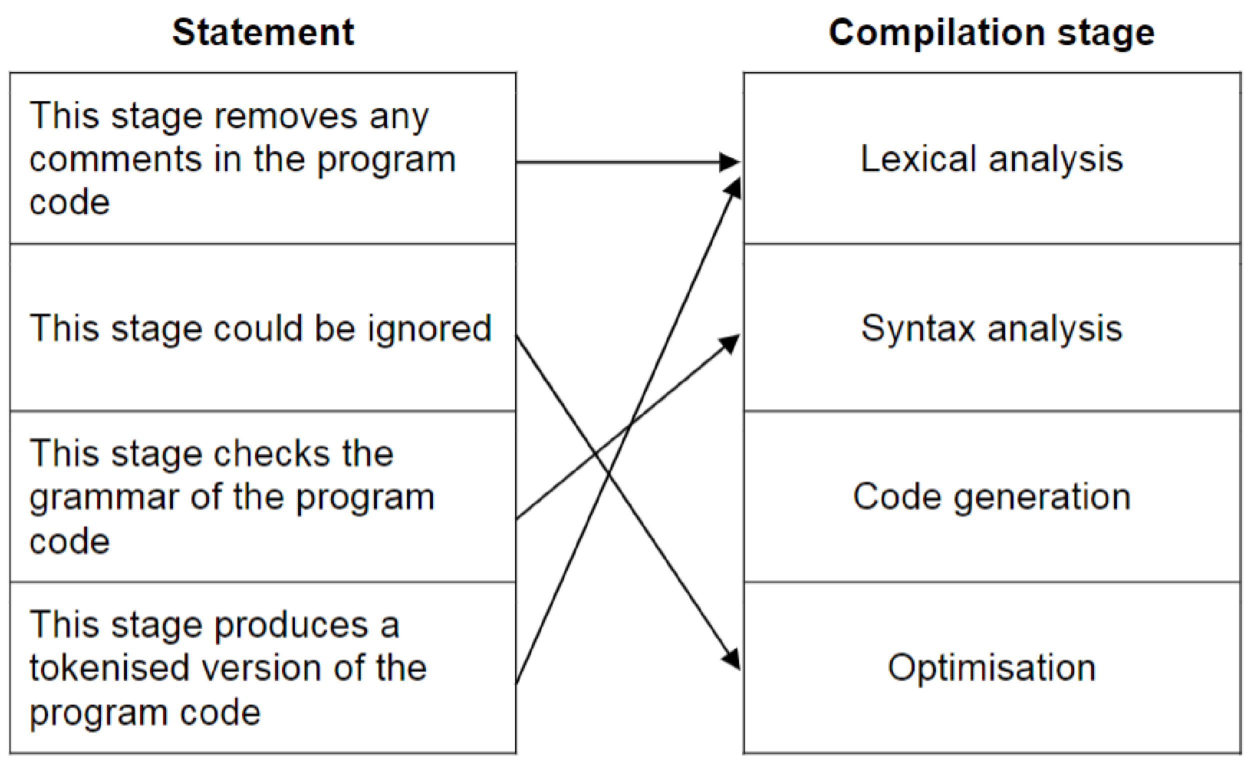
```
01  LDD 436 //loads value A
02  ADD 437 //adds value B
03  STO 612 //stores result in X
04  LDD 436 //loads value A
05  ADD 437 //adds value B
06  ADD 438 //adds value C
07  STO 613 //stores result in Y
```

Suggest what a compiler could do to optimise this code.

- Remove the second instances of LDD 436 // remove line 04
- Remove the second instance s of ADD 437 // remove line 05
- The value required is already stored in the accumulator

Why compiled version helps protect the security of the source code

- Compiler produces executable version - not readable
- Difficult for reverse engineering


Statement
Compilation stage
This stage removes any comments in the program code
This stage could be ignored
This stage checks the grammar of the program code
This stage produces a tokenised version of the program code
Lexical analysis
Syntax analysis
Code generation
Optimisation