Types of programming languages and Translators

What is source code?	• A program written in a high level language ...by the user • Easy for people to understand • It cannot run until it has been • ...translated by the translator (compiler/interpreter/assembler) to object code
What is object code?	• Low level/machine code/binary code • ...used by computer • Produced by the translator
What is executable code?	• A complete program • ...that the computer can run • ... without further translation • Now in machine code
What is intermediate code?	• Code that has been translated usually by a compiler • ...into code that needs to be further translated (usually interpreted) before it can be run • Can run on a variety of computers • Same intermediate code can be obtained from different high level languages. • Improves portability • Requires an interpreter and virtual machine to run
What does the linker do?	• Combines modules/library routines • ...that are already compiled (are in object code)
What does the loader do?	• Copies modules into memory • ...from backing store • ...ready for execution • Completes address links to the program
What does the linker do?	• Combines modules/library routines • ...that are already compiled (are in object code)
What does the loader do?	• Copies modules into memory • ...from backing store • ...ready for execution • Completes address links to the program
What is an ‘assembler‘?	• A program that translates assembly code • into machine code/object code

High -Level	Low-level
• It uses **English-like statements** which can be easily read by programmers. • Designed for quick programming. • Can be translated for **multiple machine architectures.** • Need to rely on compiler to optimise the code. • HLL may produce multiple lines of machine code per line of code // one-to-many	• Microprocessor/CPU/**Machine specific** and can **control the hardware directly.** • Can be highly optimised to make **efficient use** of the hardware and execute more quickly. • Each line of code is one instruction only • Hard to read and learn. • Only works for one type of machine architecture • Use mnemonics. - Sequence of letters and easy for a person to remember (ADD, LOAD, SUB, INP etc)

One line of Python code results in **many lines of machine code** due to the steps necessary to perform the operation in hardware

Python code
a = 5 + 10

Resulting Assembly Code
1. MOV R1, #5   ; Move constant 5 into register R1
2. MOV R2, #10   ; Move constant 10 into register R2
3. ADD R3, R1, R2 ; Add R1 and R2, store result in R3
4. MOV [a], R3   ; Store the result from R3 into the memory location of a

How an assembler work

- Reserves storage for instructions & data
- Replaces mnemonic opcodes by machine codes
- Replaces symbolic addresses by numeric addresses, for example When the assembler processes the code, it replaces the symbolic labels (like DATA) with actual memory addresses.
- Creates symbol table - is a data structure created by the assembler to map **symbolic labels** to **numeric memory addresses**.
- Checks syntax
- Error diagnostics

```
        INP       ; Read input into the accumulator
        ADD DATA  ; Add the value stored at DATA
        STA DATA  ; Store the result back in DATA
        HLT       ; Halt execution

DATA    DAT 10    ; Store initial value 10
```

```
INP       ; Read input into the accumulator
ADD 00010000; Add the value stored at 00010000
STA 00010000; Store the result back in 00010000
HLT       ; Halt execution
```

Symbol table

Symbol	Address
DATA	00010000

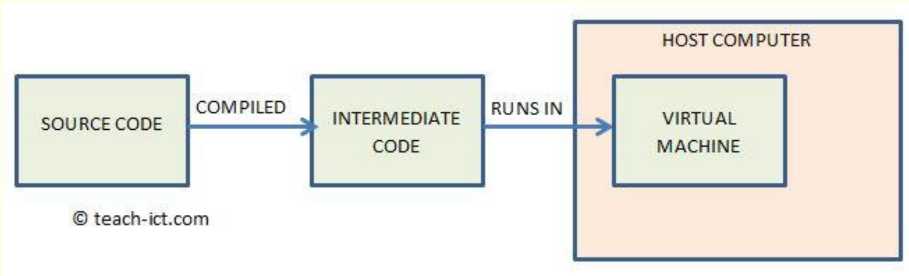
Compilers	**Interpreters**
Translate **entire source** code all in one go into Machine Code and creates an executable file (exe file)	Translate and execute source code **Line by Line**
Object code is faster processing	Slower processing
Reports all errors at the end	Reports errors as they occur
Whole program must be loaded into memory	Memory is only needed as each line of code is run
Used at the end of development (ready for shipping)	Used for development (aide debugging)
Source code hidden so it protects the code from being copied.	Access source code
Machine dependant only run on a computer with a particular machine architecture.	Platform independent as executes the source code just needs a suitable interpreter

High Level Languages and **Intermediate code:**

Intermediate code is a type of code that sits **between high-level source code** (like Python, Java, C#) and **machine code** (binary code executed by the CPU). It is **not specific to any machine**, but it's more structured and lower-level than the original source code.

Interpreter or Virutal Machine handles machine-specific execution

- Portable/ can be used on any machine - Write once, run anywhere. The same intermediate code can run on different systems using the appropriate interpreter or virtual machine.
- **Intermediate code** is a machine-independent
- Protects the source code from being copied



The instruction is split into an **Opcode** and an **Operand**
Opcode is which operation to carry out.
The **operand** specifies the data that needs to be acted on.

Mnemonic	Instruction	Alternative mnemonics accepted
ADD	Add	
SUB	Subtract	
STA	Store	STO
LDA	Load	LOAD
BRA	Branch always	BR
BRZ	Branch if zero	BZ
BRP	Branch if positive	BP
INP	Input	IN, INPUT
OUT	Output	
HLT	End program	COB, END
DAT	Data location	

Example 1: Add two numbers

```
INP;     // Input the first number
STA 90;  // Store the first number in memory location 90
INP;     // Input the second number
ADD 90;  // Add the number in memory location 90 to the accumulator
OUT;     // Output the result
HLT;     // End the program
DAT;     // Memory location 90 for storing data
```

Addressing modes

0	LDA &7
1	ADD #4
2	OUT
3	HLT
4	6
5	2
6	10
7	15
8	16
9	17

Fig. 3.1

Immediate Addressing (#)

before a number means to use the **value itself**, not the address.

Example: ADD #4 means "Load the value 4 directly into the accumulator."

Direct Addressing (default, no symbol)

The operand refers directly to a memory address.

Example: LDA 10 means "Load the value at memory address 10."

Indirect Addressing (&)

& means the operand is the address **of a memory location that contains another address**.

Example: LDA &7 means:

Go to memory address 7, read the number there 15

Then go to memory address 15 and load that value.

Indexed addressing

Indexed addressing uses a **base address + the value in an index register** to compute the final memory address. This allows for **array-like access** or **looping through data**.

For example:

- The instruction is **LDA 20,X**
- The index register X contains the value **0**

Then: The LMC loads the value from address **20 + 0 = 20**
So it accesses **memory address 20**.

After the operation, the index register **X increments by 1**.

How compilers work

Stage 1 Lexical Analysis

Step 1: **Lexer** scans the source code, letter by letter

Step 2 Words/**lexeme** are identified when there is a white space, operator symbol or special symbol.

Step 3 Will strip out the comments and will remove spaces in lines of code

Step 4 The Lexeme is checked to see if it is a **valid token**

Step 5 The lexeme is then stored along with the token, for example:

[keyword: if][Separator:(][Identifier: x][Operator:>] and added to a **symbols table.**

All keywords, constants and identifiers (e.g. variable names) used in the source code are replaced by '**tokens**'

Token class	Example
Identifier	*Any function or variable name*
Keyword	As If Else EndIf Function EndFunction Return
Separator	( ) &
Operator	+ - * / % ^ DIV MOD < <= > >=
Literal	Hello world
Number	-4 0 3.4
Quote	" "
Bool	True False
Datatype	Integer Decimal String Boolean

The symbol table

The symbol table plays a central role in the compilation process. It will contain an entry for every keyword (reserved word) and identifier in the program.

```
'Function to return if a student has passed an exam

Function checkScore (score As Integer)

    If score > 75 Then
        Return "Pass"
    Else
        Return "Fail"
    EndIf

EndFunction
```

The table will show:

- **Identifier or keyword**
- **Kind of item** (variable, array, procedure, keyword etc.)
- **Type** of item (integer, real, char etc.) – added during the syntax stage

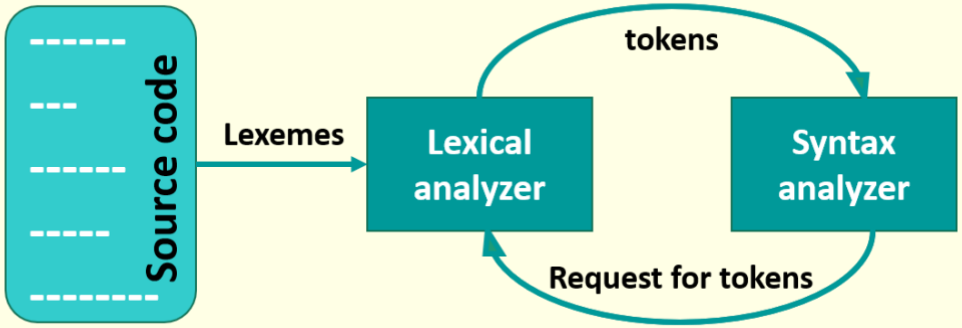
Also it will store:

- **Run-time address** of the item, or its value if it is a **constant**
- **Pointer** to accessing information (e.g. for an array, the bounds of the array, or for a procedure, information about each of the parameters).

Index	Token	Token class	Data type
0	Function	Keyword	
1	checkScore	Identifier	
2	(	Separator	
3	score	Identifier	
4	As	Keyword	
5	Integer	Datatype	
6	)	Separator	
7	If	Keyword	
8	>	Operator	
9	75	Literal	*This is added at the syntax analysis stage.*
10	Then	Keyword	
11	Return	Keyword	
12	"	Quote	
13	Pass	Literal	
14	Else	Keyword	
15	Fail	Literal	
16	EndIf	Keyword	
17	EndFunction	Keyword	

Stage 2 Syntax analysis

- This stage analyses the syntax of the statements to ensure they **conform to the rules of grammar for the computer language** in question.
- The purpose of syntax analysis or parsing is to check that we have a **valid sequence of tokens**.
- Tokens are a valid sequence of symbols, keywords, identifiers, etc.



1. **Token stream** sent to the **syntax analyser**
2. Checks that the token stream follows the grammar rules of the language
3. Stacks will be used to check, for example, that brackets are correctly paired.
4. If fails Report errors
5. Reports diagnostics
6. Create **abstract syntax tree**
7. Update identifiers in the symbols table

Stage 3 Code generation and optimisation

This is the final phase of compilation creates **object code**
To make the program run faster/ code is more efficient
To make the program use fewer resources/less memory

The disadvantages of code optimisation are:
• it will increase compilation time, sometimes quite considerably
• it may sometimes produce unexpected results.

Example:

- Removes variables and subprograms that are not used
- Removing lines of code that are never accessed

What is a linker?

- Combines/links code/programs to files/software libraries…
- …to form a single executable file
- **Static linkers** combine code and libraries into one file
- **Dynamic linkers** link - add addresses to libraries

What is a loader?

- It is part of the operating system
- Loads an **executable file** (into memory)...
- …from secondary storage
- Loads the required **software libraries**

Libraries

Library programs are ready-compiled programs, grouped in software libraries, which can be loaded and run when required. In Windows these often have a .dll extension.
Most compiled languages have their own libraries of pre-written functions which can be invoked in a defined manner from within the user's program.

Advantages of library routines
These libraries can be imported into a user's program and have many advantages including:
• They are tested and error-free
• They save the programmer time in "re-inventing the wheel" to write code themselves to perform common tasks