

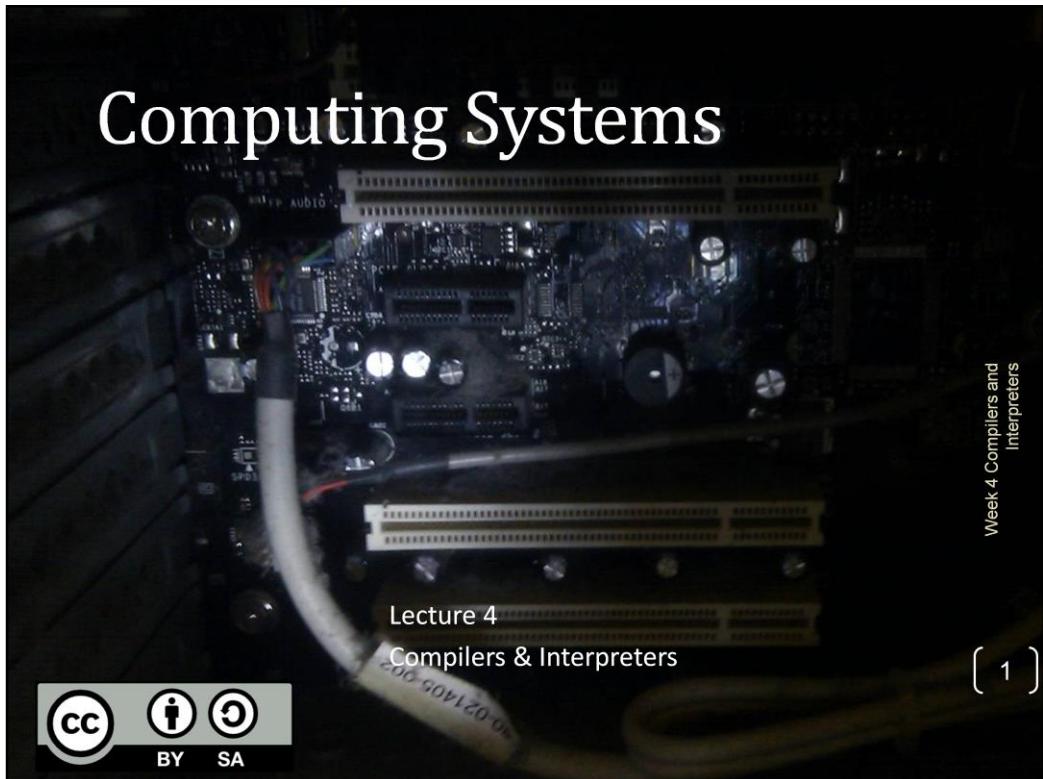
Computing Systems

Week 4 Compilers and
Interpreters

Lecture 4

Compilers & Interpreters

[1]



Zeroes and Ones (Again, again, again)

- CPU processes instructions in machine code
 - Each instruction is a pattern of 1's and 0's
 - Not easy for humans to read or write!
- Assembly Languages
 - Use mnemonic names for each machine code instruction
 - E.g.: represents **00000101** as **DEC B**
(decrease the number stored in **register B** by one, an instruction for the Z80 processor)
- Will return to Assembly later

The CPU processes instructions in Machine Code.

The transistors, the key components which make up the CPU, only understand ON / OFF.

All of the instructions the CPU processes are patterns of 0 / 1s, which are not easy for humans to read or write. Earliest programs were written in machine code.

People write using mnemonics which are easier than binary. The assembler can turn turn mnemonics into binary.

Programming In Assembly

- Programming in **assembly** is possible
 - But *difficult & tied* to particular CPU designs
 - And very *slow*: very *simple operations* may require *dozens of lines of assembly code*
- Hand-crafted assembly **can** be very efficient
 - Generally not worth time & effort for the gain
 - DOOM (1993) was one of the last major games to use significant amounts of hand-crafted assembly programming

Quake also used assembly to program games.

High Level Programming Languages

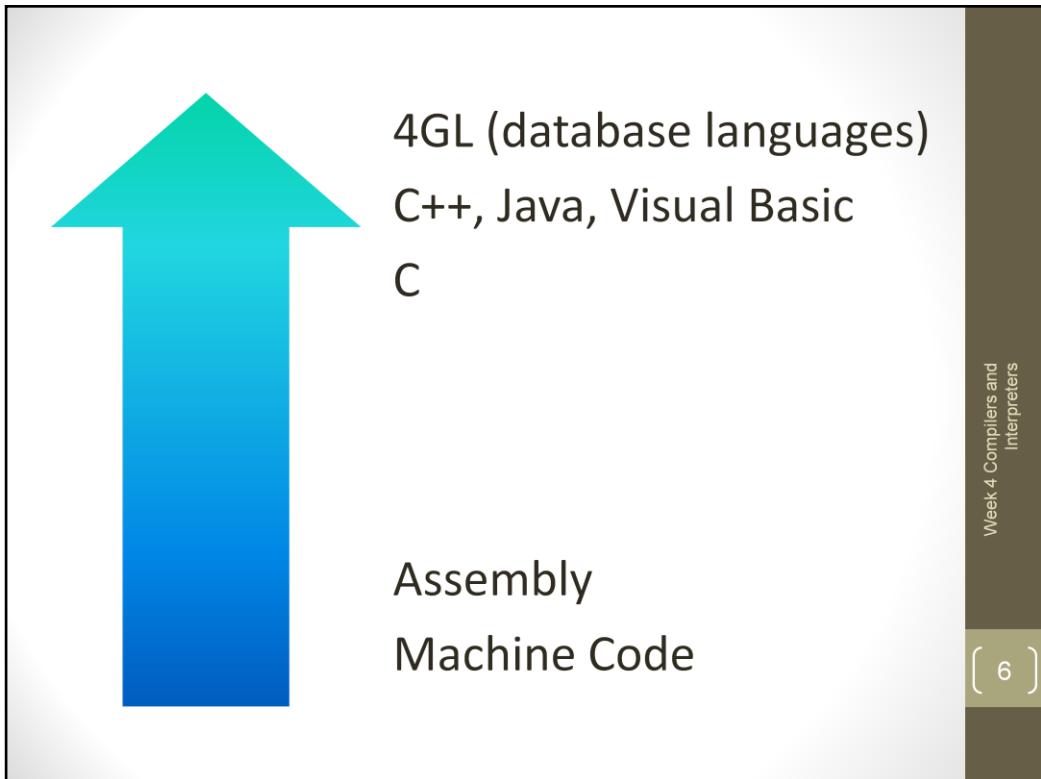
- **Abstracted** away from the low-level hardware
 - More **readable**
 - **Portable** – a program developed in a high-level language may be used on different **CPU architectures**
 - More **powerful**: a single instruction in a high-level programming language may equate to *many lines of assembly or machine code*
- **HUGE** range of high-level programming **languages** exist

Can write the program once then compile to run on different computers with different CPUs in them.

What Programming languages have
you used?

Week 4 Compilers and
Interpreters

(5)



High level and low level programming.

Assembly and Machine code are low level languages are closer to the processor.

C++ etc are high level language.

C is considered a lower level language as it allows a lot of direct access to the underlying hardware, and has some lower level language features.

Language Translators

- Software that **converts** a program from one *form* to *another*
 - Most commonly from a *high-level* (**human generated**) form to a *low-level* (**machine code**) form for the computer to use
- The **high level language** code usually exists in text files known as the **source code**

Translator converts source code into machine code.

Execution Models

Models for how a program is turned from high-level code to machine code for running on a computer

- **Interpreted Languages**
 - E.g. BASIC, scripting languages
- **Compiled Languages**
 - E.g. C, C++
- **Translated Languages**
 - Code is converted to into another programming language for *interpretation* or *compilation*

Translator can take a source code and convert into C for example from another programming language..

Languages that are usually interpreted:

**BASIC (most versions), Javascript, Lua, PhP,
Python, Perl, Game Maker, Ruby, ...**

Languages that are usually compiled:

**Algol, C, C++, Objective-C, Pascal, Ada, C#,
COBOL, Delphi, Eiffel, Common Lisp, Forth,
Java, Scheme, Smalltalk, ML, OCaml, Visual
Basic, ...**

Interpreters

```
10 REM A simple BASIC program
20 PRINT "Hello World!"
30 GOTO 10
```

- When this **BASIC** program is run, the interpreter **converts** each line in turn into machine code
- Conversion to machine code occurs ***while the program is running***

[10]

Each line is converted from source code to machine code each time the program is run.

Advantages are: the code is often more flexible and can be changed at any time. But it is slower as some line e.g. the ones inside a loop are interpreted many times.

Language Interpretation

- Can **change** source code at any time
- **Translation of instructions** while program is **running** slows down the program
- Lines inside a **loop** may be **translated many times**
- Availability of the source makes it easy for others to **reverse engineer** your program
 - Discover how your program works to make their own versions

Flexible, easier to develop and test.

Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.

Compilers

```
#include <stdio.h>
int main() {
    printf("Hello World");
}
```

- Before this **C** program can be run, a **compiler** must be used to *convert the whole program* into **machine code**
- Conversion to **machine code** occurs some time **before** the program is run

```
#include <stdio.h>
```

The pre-processor operates on this statement. It tells it that it needs to take the contents of the file stdio.h and adds it into the source code before compiling.

C/C++ Compilation

- First **source code** files are **pre-processed**
 - Pre-processor instructions (**directives**) modify the **source code** itself before compilation
- Next, the source code files are **compiled**
 - A single project may have *dozens* or *hundreds* of **different source code files**
 - The output is an **object file** for each **source** file
- Finally, a **linker** combines all of the object files into the **executable** program file

Each source code file that is compiled (may have dozens or hundreds) creates an object file (in machine code).

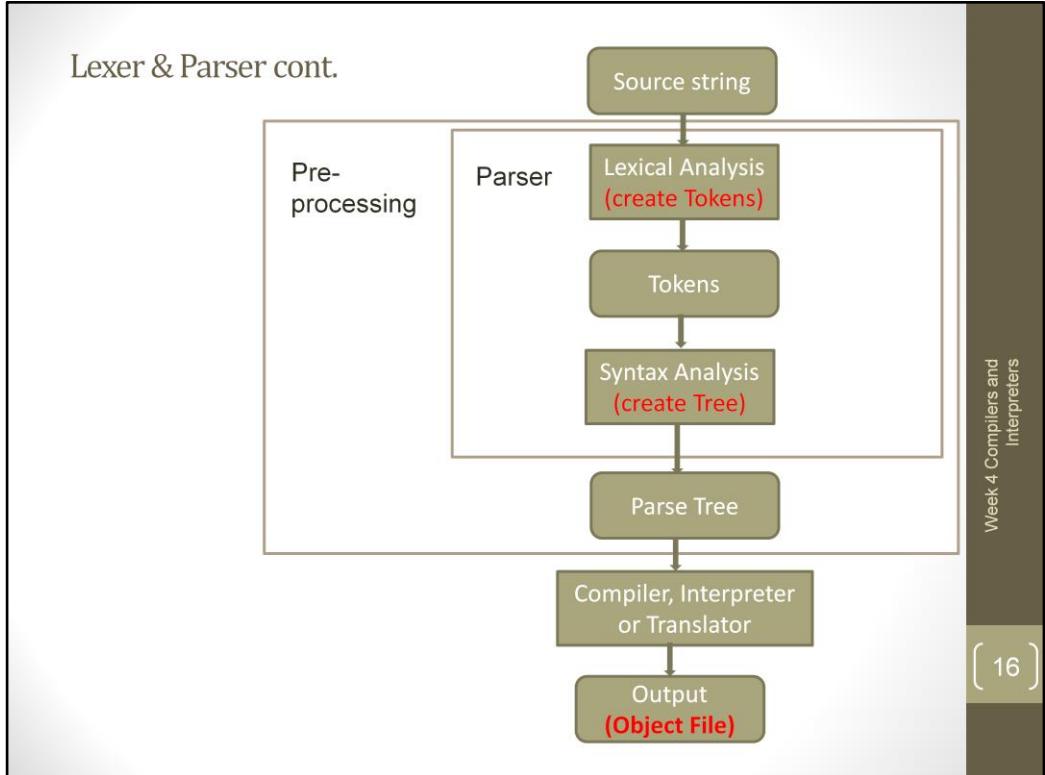
Compilation has two phases.

Lexer & Parser

- **Lexer (lexical analyser)** reads **source code**
 - Replaces **keywords** with **tokens**(small pieces of source code: keyword(name of a variable, function, or label); identifier or symbol name)
 - Identifies **variables** and **data**
 - Discards **comments, spaces**
- **Parser** creates **syntax tree** from **tokens**
 - **Syntax errors** detected here: where programs don't follow the rules of the language
- **Syntax tree** fed to a **code generator** stage which produces **machine code**
 - Possibly followed by additional optimizer stage

Lexer inputs into the Parser then Parser feeds into a Code Generator.

Lexer & Parser cont.



Lexical Analysis Tokens example

if(X > 3.1) { printf ...



Character Stream

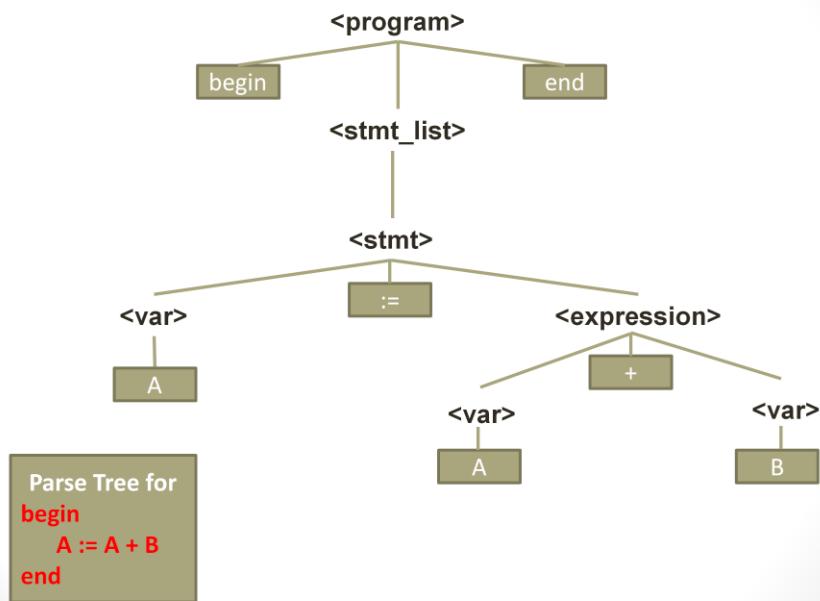
Lexical Analyzer



Token Stream

KEYWORD “if”	BRACKET-R “{”	IDENTIFIER “x”	OPERATOR “>”	NUMBER “3.1”
-----------------	------------------	-------------------	-----------------	-----------------

Parse Tree example



Linker

- Combines **object** files to create **executable program**
- Can also create ***dynamic link libraries – libraries*** of *executable* code to be used by other programs
- Creates and uses **lib** files – libraries of **compiled code** for use in **compiling other programs**

[19]

When we compile a file we end up with an object file.

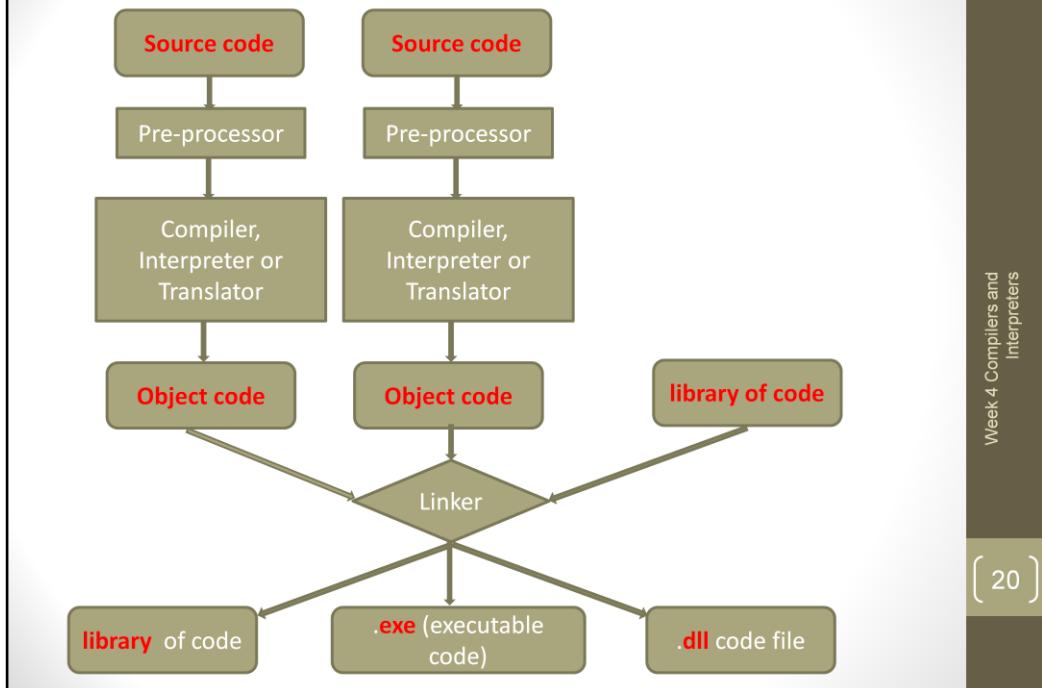
The linker combines the Library (Lib, pre-existing object files) files with the Object files.

Produces program output – exe an executable file(can run itself), an object file(lib) stored in the library. lib files are compiled code that can be used in the linking stage of compiling a new project.

It is mainly used for creating code libraries for third party use - and means that the code library itself does not need to be recompiled every time it is used, it is added at the linking stage instead.

dll files, in contrast, are used at run-time, and allow third party libraries to be used in a different way - instead of being 'statically linked' into the project (and being included in the executable file), the library code is kept separate. This can be particularly effective if the library is very commonly used - instead of having the code added to every project that uses it, the DLL only needs to be installed once. Additionally, some open source projects allow commercial use only where added as dynamic link libraries.

Linker cont.



Comparison of compiled and interpreted code

- Using a **compiler** separates *translation* and *execution* of a program.
 - In contrast of an *interpreted* program the *source code* is **translated** only **once**.
 - The *object code* is *machine-dependent* meaning that the *compiled* program can **only** be **executed** on a **machine** for which it has been **compiled**
- An **interpreted** program is **not machine-dependent** because the *machine-dependent* part is in the interpreter itself.
 - A pure interpreter **reads** the source text of a program, **analyzes** it, and **executes** it as it goes. This is usually **very slow**--the interpreter spends a lot of time analyzing strings of characters to figure out what they mean.
 - A pure interpreter must **recognize** and **analyze** each **expression** in the **source** text **each time** it is **encountered**, so that it knows what to do next.

Portable Programs

- During compilation a program is translated from a *high level portable* form to a *low level CPU* specific form
 - The same source code can be **compiled** for a **range of different machines** (often requiring *some changes!*)
- Many applications need the relative **speed** advantage of **compilation** while retaining a **high** degree of **portability**

[22]

Pre-processor directives can be used to make changes to code e.g. **#include**. e.g. depending on compiling for a Mac or Windows machine, different Libraries might be included. i.e. we do not need to recompile for a different machine.

Bytecode and Virtual Machines

- Solution may be *intermediate compilation*
 - A high-level language is compiled into a **machine-code** like **byte-code**
 - The **byte-code** is executed by a ‘**virtual machine**’ (VM)
 - **Virtual Machine(VM) → emulates a range of different processors**, running a set of instructions which should be able to run on a range of machines.
 - The VM **interprets** code, translating **byte-code** to **machine code**
 - **Java** is probably the most well known example.

Byte code is executed on a ‘**virtual machine**’ (VM)
→ **emulates a range of different processors**,
running a set of instructions which should be able
to run on a range of machines. Giving better
portability.

Java is a portable programming language,
therefore Java can run on any computer as long
as a virtual machine is available. Therefore it is
more efficient.

Intermediate Compilation: Java

- Java is **not** the same as **JavaScript**!
 - Javascript → Scripting language running on a web server or in a **web browser**. (e.g. Cookies Alert)
- **Byte-code** is relatively **low-level**
 - Does **not** need to be **human readable**
 - Allows the **VM** to execute **byte-code faster** than traditional **interpreted** code
- **VMs** can be developed for different platforms
 - So the same **Java** program can run on **Mac, Windows, ...**
- **Just-In-Time compilation** to improve performance

Java → programming language which gets compiled to byte – code which runs on a virtual machine, byte code is relatively low level and does not need to be human readable.

Javascript → Scripting language running on a web server or in a web browser.

Just-In-Time → allows processor to convert a command to machine code but it remembers the compiled version, therefore in a loop it only needs compiled once.

Common Intermediate Language

- **CIL**

- Part of Microsoft .Net framework
- **CPU & Platform** independent
- Range of languages compile to **CIL**
- **Bytecode** can be translated into **native code** OR run by a **VM**
- Virtual Machines can be used to test code on a wide range of CPU architectures.
 - Think of a simple app designed to run on a wide variety of mobile devices.
- **VM** also uses **Just-In-Time** compilation
 - First time a command is **interpreted**, it is **compiled**
 - *Repeats of a loop* benefit from **speed up**

CIL → e.g. C-sharp and XNA, can use XNA to develop a game which can run on Windows, running on an Intel CPU, then run the same code to run the same game on an x-box, which uses a different processor architecture and has a different set of low level command which it understands.

Create Your Own

- Developers continue to create new programming languages
 - To exploit *new hardware capabilities* or computer science theories
 - To improve *software development* efficiency
 - To improve on *existing languages*
 - To prove a *theoretical* point
 - For *fun*

A virtual machine lets you use more portable code. But you could develop your own.

LOLCODE (yes really)



HAI → program start

KTHXBYE → program end

What does this program do?

HAI
CAN HAS STDIO?
I HAS A VAR
IM IN YR LOOP

“I HAS A” used to declare a variable

UP VAR!!1

Increment variable by one

VISIBLE VAR

IZ VAR BIGGER THAN 10? KTHX

IM OUTTA YR LOOP
KTHXBYE

This loop will repeat until VAR = 11

Loop to print numbers 1 to 10 – though code inspection makes it look like it will print numbers 1-11

Worth checking out...

Writing A Compiler

- The first compilers had to be written in **assembly**
 - There were **no compilers** to use!
- Can use any **high-level language**
- Range of tools exist to make it easier to write **compilers**
 - Lex and **yacc**(Yet Another Compiler Compiler) for writing **lexer** and **parser**
 - Lots more at **compilertools.net**

Lex → Lexer

yacc → Parser, Yet Another Compiler Compiler

Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.

Programming Paradigms

- Over the past half century a number of different **programming paradigms** have emerged
- Increasing **complexity** and **size** of software systems required changes to the manner of *software construction*
- A **programming paradigm** is a fundamental style of computer programming, a way of building the structure and elements of computer programs.

[31]

Programming Paradigms cont.

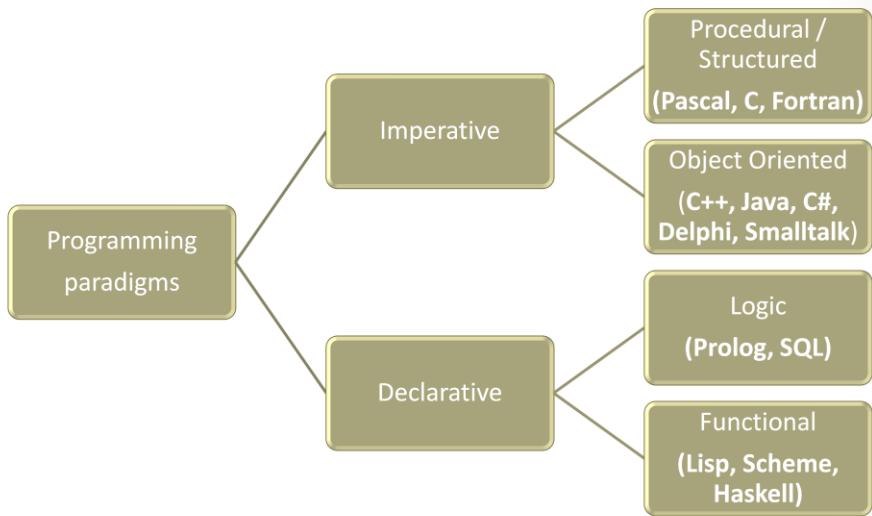
- Capabilities and styles of various programming languages are defined by their supported programming paradigms; some programming languages are designed to follow only *one paradigm*, while others support multiple *paradigms*.

Major programming paradigms

- Major programming paradigms:
 - Imperative
 - Declarative
 - Functional
 - Object-oriented
 - Logical
 - Structured / Procedural

[33]

Main programming paradigms cont.



Week 4 Compilers and
Interpreters

[34]

Imperative & Declarative

- **Imperative** Programming
 - Program is a sequence of instructions
 - Each instruction changes the program **state**
 - **Procedural (or Structured)** and **Object-Oriented** Programming
- **Declarative** Programming
 - Code declares **what** program should achieve, *not how* it should achieve it
 - **Functional or Logical** Programming

Imperative

Procedural → C, Pascal

Object-Oriented → C++, C sharp, Java

Declarative → Functional, Logical

Structured Programming

- Examples: **C, Pascal**
- **Program** can contain *blocks* of code known as **functions or procedures**
- Statements may *call* other **functions or procedures**
 - Execution jumps to called procedure & returns to just after calling statement when procedure is completed

```
int GetMax(int a, int b) {  
    if (a > b)  
        return a;  
    return b;  
}
```

2. Execution jumps to here

3. Return statement ends a function

```
main() {  
    ...  
    GetMax(input1, input2);  
    ...  
}
```

1. Program executes this statement

4. And execution continues here

Start at Main. **GetMax** → is a calling statement

Object Oriented Programming

- Dominant programming paradigm?
- Examples: **C++, Java, C#, Delphi, Smalltalk**
- Groups data with related functions into *classes* and *objects*
 - Classes **encapsulate** design: to use a class, there is no need to understand how it is implemented
- **Objects** can communicate with other **objects** (*message passing*) & it is possible to derive new classes from existing ones (*inheritance*)

Functional Programming

- Examples: **Lisp, Scheme, Haskell**
- Emphasises functions where **imperative programs** emphasise program **state**
 - In a functional programming language, a function's result depends only on the **input parameters, not on the program state**
- May be **less efficient in memory & CPU** use
- Relatively **limited** use outside academia

[39]

Logical Programming

- Examples: **Prolog** and **variants**
- Based on propositional logic
- Program is given a knowledge base or *rules* and *facts*
 - Example fact:
`father_child(mike, tom) .`
 - Example rule:
`parent_child(X, Y) :-
 mother_child(X, Y) .`

Mike is the Father of Tom

If x is the Mother and y is child, then x is the parent and y is the child.

Prolog Example

```
mother_child(trude, sally).  
father_child(tom, sally).  
father_child(tom, erica).  
father_child(mike, tom).  
  
sibling(X, Y) :- parent_child(Z, X),  
    parent_child(Z, Y).  
parent_child(X, Y) :- father_child(X, Y).  
parent_child(X, Y) :- mother_child(X, Y).  
  
?- sibling(sally, erica).  
Yes
```

Two siblings with same parents therefore sally and erica are siblings. Program is a collection of rules or facts.

Others

- Event Based
- Parallel
- Agent Oriented
- Many other paradigms exist
 - Some languages are examples of many different paradigms or may have extensions applied to enable other paradigms

[42]

Event Based → C, C++

Parallel → multiple things can happen at the same time.

Quick quiz

- Join the ‘Socrative’ app ‘Room 642124’ and try the quick quiz.

Further Reading

- Wikipedia
 - Compiler, High level language, List of programming languages, LOLCODE, ...
- Compilertools.net
- Programming Languages and Compilers
 - [http://www-
inst.eecs.berkeley.edu/~cs164/sp11/](http://www-inst.eecs.berkeley.edu/~cs164/sp11/)
 - A complete course on the construction of programming languages – quite advanced

Next Week

- ***Required Reading*** – HCW:
 - RAM p56-59
 - Data Storage, p148-155, p158-161, 167-171, 177-179, 185-193
- Additional Reading
 - HCW – Remaining sections on Data Storage
 - PCH – Chapter 12, Computer Memory



This work by Daniel Livingstone at the University of the West of Scotland is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Credits:

Title Image, CC-BY-SA Daniel Livingstone

LOLCODE examples from: <http://en.wikipedia.org/wiki/LOLCODE> CC-BY-SA

Prolog example from <http://en.wikipedia.org/wiki/Prolog> CC-BY-SA

Product names, logos, brands, and other trademarks featured or referred to within the these notes are the property of their respective trademark holders. These notes have been produced without affiliation, sponsorship or endorsement from the trademark holders.