1-2-2 Applications Generation

Revision sheet

1 The Nature of Applications

There are to broad categories of software - systems software and applications software.

1.1 Systems Software

This is the software needed to run the computer's hardware and application programs. This includes the operating system, utility programs, libraries and translators.

1.1.1 Operating System

This is a set of programs that lies between applications software and the hardware of the computer. It has a number of different functions including: resource management and providing a user interface.

1.1.2 Utility Programs

This is a category of systems software which is designed to optimise the performance of the computer of perform tasks such as managing and compressing/decompressing data and files and providing firewall. There are a number of different types

- Disk defragmentation;
- Automatic backup;
- Automatic updating;
- Virus checker;
- Compression software.

1.2 Applications Software

This can be categorised into two categories.

1.2.1 General-Purpose software

This is software which can be used for a variety of tasks. Examples include: word-processors, spreadsheet or graphics packages.

1.2.2 Special-Purpose Software

This performs a single specific task or set of tasks. Examples include: restaurant reservation managers, web browsers and media players. There are two different types of special-purpose software: *off the shelf* and *bespoke*. Each have their own pros and cons.

Off The Shelf	**Bespoke**
Less expensive as cost is shared between multiple license holders	Expensive and requires experts to design solution
May contain some unwanted features and some less-essential features	Features are customised to the client, more features added as they are needed
Ready to deploy immediately	Can take a long time to develop
Well documented, tested and error free	May contain errors which don't get shown immediately

Table 1: Comparison of Off The Shelf and Bespoke software

2 Sources of Software

There are two main sources of software, open source and closed source.

2.1 Open Source

Open source software is governed by the *Open Source Initiative* that says: there is no charge for the software, anyone can use it; open source software must be distributed with the source code so anyone can use it; developers can sell the software that they have created; any new iterations or branches from the original open source project must also be open. Usually there is community based tech support available, with minimal support made available from the original developer.

2.2 Closed Source

Closed sourced (proprietary) software is usually sold, with a license required to use it. There will be restrictions on how the software is used; for example, a single license may only allow one concurrent user or may permit up to 25 users on one site. The company or person who wrote the software will hold the copyright. Users will not have access to the source code and will not be allowed to modify the package. There is also usually tech support available from the developer/company.

3 Programming Language Translation

Humans write code in whatever language fits the project they are writing. This high level language needs to be converted into a low level language which can be run by whatever device it is intended to be run on. This low level language which machines can run is called machine code and is written in pure binary (but is often displayed in hex). There are three different translators which can be used: Compilers, Interpreters and Assemblers.

3.1 Assembler

Assemblers translate assembly code to machine code. Assembly language is also a low level language which makes use of mnemonics for each command. The assembler assembles the entire program at once, this means once assembled, the program can be run again and again.

3.2 Interpreters

Interpreters translate and run source code into machine code line by line, this means they are very good for pinpointing errors as they will be shown as soon as they are encountered. It does not create an executable file at the end of translation so every time the program is run, it will have to be interpreted which makes it slow at runtime. There is quite a significant disadvantage to compilers - each statement has to be translated into machine code every time it is encountered.

3.2.1 Bytecode

This is an intermediary stage between an interpreted language and machine code. In some cases, *Java*, the whole program is compiled into Bytecode first then run within a virtual machine (*Java Virtual Machine - JVM*). Within the virtual machine, the bytecode is interpreted. Using bytecode allows for greater platform compatibility as the virtual machine is the thing interfacing with the operating system, not the individual applications.

3.3 Compilation

Compilers translate the whole program 'at once' into machine code. A compiler may scan through the source code multiple times before producing object code (machine code which is missing bits that Linkers fill in). Compiling may take some time, but once complete, it produces an executable file which can be run without the need to compile again. Errors are listed in the error report which is produced once the full program has been compiled. These errors can only be fixed by changing the original source code and compiling the program again.

3.4 Comparison

Compilers have many advantages over interpreters: the object code can be saved to disk and run whenever the program is needed, without having to be recompiled every time; the object code is faster than interpreted code; the object code produced by a compiler can be distributed or executed without having to have a compiler present; the object code is more secure (a lot of effort would be required to reverse engineer it). Interpreters also have advantages over compilers: platform independence; quicker runtime loading as there is no lengthy compilation required every time it is run.

4 Stages of Compilation

There are three stages of compilation.

4.1 Lexical Analysis

1. Extra spaces are removed
2. All comments are removed
3. Simple error checking
   - illegal identifiers would be flagged as an error
   - assignments of illegal values to data types would be flagged as an error
4. All elements are identified then tokenised

5. The variable names / identifiers are put in the symbol table.

Tokens are a string of binary digits of a fixed length. The symbol table lists each identified token and what type of symbol it is.

4.2 Syntax Analysis

1. Accepts the output from lexical Analysis

2. Statements and arithmetic expressions/tokens are checked against the rules of the language (defined in Backus-Naur Form), for example are all the brackets matched and used correctly.

3. Produce a list of errors which are outputted at the end of compilation. If there are no errors, the code is passed to code generation as an *abstract syntax tree* with any additional details added (eg. data types, scopes, addresses).

4.2.1 Semantic Analysis

Syntax analysis also includes semantic analysis. This is where the compiler identifies if *identifiers are used appropriately throughout the whole program.* For example: variable not declared multiple times; variable assigned before referenced; assignment compatible with declared type; operations on variables compatible with type.

4.3 Code Generation

This is the last phase of compilation. It produces machine code/executable/intermediate code which is equivalent to the source program. During this phase, variables and constants are given addresses and relative addresses are calculated. The code can also be optimised here.

4.3.1 Optimisation

This makes the code as efficient as possible. It increases processing speed, generally by reducing the number of instruction; programmers can choose this however, either increasing speed or reducing size. It can include: loop optimisation; making calculations more efficient and changing relative addresses to absolute in memory.

5 Linkers, Loaders and Libraries

Linking and loading are often done together by the same software but are considered separate processes.

5.1 Linkers

Linkers (or Link Editors) are computer programs that take one or more object files generated by a computer and compiles them into a single executable file, library file or another object file.
Dynamic Linkers: The part of an operating system that loads and links the shared libraries (DLLs) for an executable file.

5.1.1 Difference between static and dynamic linking

In static linking, the linker copies all of the library routines used in the program into the executable image. This requires more disk space and memory than dynamic linking but is both faster and more portable; it also doesn't require the presence of the library on the system where it is run. Dynamic linking places the name of the shareable library in the executable image, actual linking with the library routine doesn't occur until the image is run, when both the executable and library are placed in memory; an advantage is that multiple programs can share a single copy of the library.

5.2 Loaders

The loader is the part of the operating system that is responsible for loading programs and libraries into memory, only when dynamic linking is used. It reads the contents of the executable file containing the program instructions and carries out the tasks required for prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.