University of Portsmouth
BSc (Hons) Computer Science
Second Year

Programming Applications and Programming Languages (PAAPL)

M30205
September 2023 - June 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

Teaching Block I

Programming Applications

There are no notes for this Teaching Block. It was entirely practical with the coursework involving no written component.

Teaching Block II

Programming Languages

Page 1

Lecture - Introduction To Progrmaming Languages

2024-01-26 1400 Jaicheng

This lecture will introduce us to the many different ways in which a programming language can be categorised.

1.1 Programming Domains

A *Programming Domain* is one way to think about & categorise a programming language. We have different different programming domains for different applications as each application requires a specialised instruction set to improve efficiency for the programmer. Everything humans do can be solved by a computer, the number of programming domains reflects this.

1.1.1 Scientific Applications

Scientific Applications of a programming language would be to do mathematical operations on some data which would result in an output. This could be used in applications such as weather forecasting where data on the current weather is fed into it - and a simulation is used to simulate the future weather conditions. Scientific applications will complete a large number of floating-point computations and use arrays. An example of a language in this domain is *Fortran* (FORmula TRANslating system, created by IBM).

1.1.2 Business Applications

Business Applications are designed to be used by businesses to complete business functions. For example, batch printing payslips. They use decimal numbers and characters. An example of a language in this domain is COBOL (COmmon Business-Oriented Language).

1.1.3 Artificial Intelligence

In the *Artificial Intelligence* domain, symbols are manipulated, rather than numbers and linked lists are used. Nowadays, this domain is now more talking about reasoning, facts and truth verification. An example of a language in this domain is LISP (LISt Processing).

1.1.4 Systems Programming

Systems Programming is concerned with the control of the hardware of the computer, the management of the storage, the display control and management of other components such as peripherals. The languages used, such as C, need to be specifically designed for this domain due to the required low level interactions between the program and the hardware.

1.1.5 Web Software

Web Software is arguably one of the most popular domains in these modern times. Much of the modern software is developed as a website, for easy use across multiple devices. The languages used are eclectic and each serve a particular purpose; for example, HTML for markup, PHP for scripting and JavaScript for adding interactivity.

1.2 Language Categories

1.2.1 Machine Languages

The *Machine Language* family of languages are hardware implemented languages; which means the instruction set available within them is the instruction set available on the CPU. This means the instruction set is limited in size and will be represented as binary (or hexadecimal) numbers.

1.2.2 Assembly Languages

The *Assembly Languages* family of languages are a simplification of Machine Languages. In essence, they are the machine language with a 'human-friendly' outside layer, meaning that they are legible to most people. To be executed, they require translating to machine code (which involves the use of a translator or interpreter). They come with labelled storage locations, jump targets and subroutine starting addresses in addition to the basic Machine Language instructions.

1.2.3 High Level Language

The *High Level Language* family is another step up from Assembly Languages. Their syntax is very close to natural language syntax, making it much more legible and easier for programmers to read, write, understand and memorise. They usually will come with variables, types, subroutines, functions, the ability to handle complex expressions, control structures, and composite types. Examples include: C and Java.

1.2.4 Systems Programming Language

The *Systems Programming Language* are effectively high level languages who also deal with the low level operations. For example C, C++, and Ada. They process the memory & process management, I/O operations, device drivers, operating systems.

1.2.5 Scripting Languages

The *Scripting Languages* are a set of languages which exist to automate tasks, saving humans time. They will commonly be used to: analyse or transform a large amount of regular textual information; act as a glue between different applications; or bolt a front end onto an existing application. The languages used are often interpreted and will often include lots of string processing functions, such as in Python or PHP.

1.2.6 Domain Specific Languages

The *Domain Specific Languages* are highly specialised languages which are used in a specific area only. For example the Adobe PostScript language is used for creating vector graphics for electronic publishing.

1.3 Categories by Paradigm

There are three different categories.

1.3.1 Procedural

A program is built from one or more procedures (can also be called subroutines or functions) and the program will revolve around variables, assignment statements and iteration. Some languages will also support Object Oriented programming as well as some supporting scripting. Examples of languages include: C, Java, Perl, JavaScript, Python, Visual Basic, C++.

1.3.2 Functional

Functional languages work by applying a function to a given parameter. Languages include: Haskell,LISP, Scheme, F#, Java 8.

1.3.3 Logic

Logical rules are used to do reasoning over given facts which draws conclusions. The logical rules do not have to be defined in any particular order. Languages include: Prolog.

1.4 Categories by How Tasks are Specified

1.4.1 Imperative Languages

In imperative languages, you have to explicitly instruct the computer what it needs to do to reach the goal, computing tasks are defined as a sequence of commands which the computer performs. The program will state in step-by-step instructions what the computer needs to do. This means that the implementation of the algorithms, and therefore the efficiency of the algorithms is down to the developer. Procedural languages belong to this category.

1.4.2 Declarative Languages

In declarative languages, the computer gets told the desired results, without explicitly listing the the commands or steps which the program must undertake to reach its goal. Functional and logical programming languages belong to this category.

Page 2

Lecture - Overview and Evaluation of Programming Languages

2024-01-26 14:00 Jaicheng

2.1 The 'TPK' Algorithm

The TPK algorithm was designed by *Trabb, Pardo* and *Knuth* in the 1970s for illustration purposes. It is designed to:

1. read 11 numbers (entered by the user using their keyboard) into an array,
2. process the array in reverse order, applying a mathematical function to each value
3. then for each value - reporting the value or a message saying that the value is too large

The algorithm includes all the basic constructs which would be expected to exist in a modern language therefore making it useful to use when understanding how languages work. A pseudocode implementation of the TPK algorithm is below:

```
input 11 numbers into a sequence A
reverse sequence A
for each item in sequence A
    call a function to do an operation
    if result overflows
        alert user
    else
        print result
```

2.2 Fortran

Fortran (*For*mula *Trans*lation) is the first well-known high-level programming language. It was developed by a team at IBM led b John Backus with the goals: to lower the costs involved with programming and debugging; and to compete with "hand coded" assembly language programs in terms of execution speed. The first Fortran compiler, built for the IBM 704 mainframe, was completed in 1957.

Early source code had a strict, specific, format which was in part due to it being a punched-card program where the column and row position of the punch is important.

The TPK algorithm in Fortran is shown below:

```
C THE TPK ALGORITHM IN FORTRAN
  FUNF(T)=SQRTF(ABSF(T))+5.0*T**3
  DIMENSION A(11)
```

```
    1 FORMAT(11F12.4)
      READ 1, A
      DO 10 J=1,11
      I=11-J
      Y=FUNF(A(I+1))
      IF(400.0-Y) 4,8,8
    4 PRINT 5,I
    5 FORMAT(I10,10H TOO LARGE)
      GOTO 10
    8 PRINT 9,I,Y
    9 FORMAT(I10,F12.7)
    10 CONTINUE
      STOP
```

A letter `C` in the first column indicated that the card was a comment and as such it should be ignored by the compiler. Non-Compiler cards were divided into four fields:

1-5 is the label field; a sequence of digits here indiciates the purpoose of the card and therefore the instruction.

6 is a continuation field whereby a non-blank character here caused the card to be taken as a continuation of the statement on the previous card.

7-72 is the statement field

73-80 are ignored by the compiler. This means that they can be used for card identification purposes in the event that the cards were dropped.

The restrictions on the structure of the code were removed in Fortran 90, where it became a Free-Form language.

2.3 COBOL

COBOL (*CO*mmon *B*usiness *O*riented *L*anguage) was created at the end of the 1950s by the US Department of Defence. It was initially developed as a language for business data processing from which comes its verbose syntax that was designed with the ability for managers to be able to read it in mind. COBOL was never designed to be used as a scientific language and has many critics, where programmers felt that the verbosity of the language increased program length, not readability.

2.4 Algol

Algo (*ALGO*righmic *L*anguage) was originally designed to overcome the problems with FORTRAN in the late 1950s. Arguably, it is one f the most successful high level programming languages of the time because it was influential over the design of subsequent high level languages.

Algol 60 introduced the use of formal notation for syntax, block structure (with locally defined variables, whoop whoop), supported recursive procedures (until this point, you could do it however the languages didn't like it) and readable if and for statements. Ultimately, Algol died out with the rise in FORTRAN's popularity. The TPK algorithm in Algol is shown below.

```
begin
  comment TPK algorithm in Algol 60;
  integer i; real y; real array a[0:10];
  real procedure f(t); real t; value t;
    f := sqrt(abs(t))+5*t^3;
```

```
  for i := 0 step 1 until 10 do
    read(a[i]);
  for i := 10 step -1 until 0 do begin
    y := f(a[i]);
    if y > 400 then
      write(i, "TOO LARGE")
    else
      write(i, y);
  end
end
```

2.5 Pascal

Pascal is a direct descendant of Algol, which was intended to be more efficient in order to compete with FORTRAN as a general purpose language. An early Pascal compiler was designed to be portable, compiling the source code to a virtual machine (*P-Code*). Pascal was popularised in the late 1970s as a good teaching language as it enforced a "good" programming style, it was especially popular amongst Universities. Pascal is still in development, with more recent versions adding modules and classes (for example, the Object Pascal Language, which is sometimes known as Delphi). The TPK algorithm in Pascal is shown below:

```
program example(input, output); (* TPK alg in Pascal *)
var i : integer; y : real; a : array [0..10] of real;
function f(t : real) : real;
begin
  f := sqrt(abs(t)) + 5*t*t*t
end;
begin
  for i := 0 to 10 do read(a[i]);
    for i := 10 downto 0 do
      begin
        y := f(a[i]);
        if y > 400 then
          writeln(i,' TOO LARGE')
        else
          writeln(i, y)
      end
end.
```

2.6 Systems Programming: C

In the early 1970s, it was decided that a more efficient language would be required for systems programming (such as compilers, operating systems, etc) as the languages created during the 1960s (Fortran and COBOL) weren't sufficient. C was developed alongside the UNIX operating system at Bell Labs; and has proven to be a very successful in systems programming and as a general-purpose programming language. It combines high-level features, such as functions & loops with low-level operations, such as arithmetic on memory addresses. Critics argue that C code is less readable and it's weak typing causes problems. Shown below is the TPK algorithm written in C.

```
#include <stdio.h>/* TPK algorithm in ANSI C */
#include <math.h>
double f(double t) {
  return sqrt(fabs(t)) + 5*pow(t,3);
}
```

```
main() {
  int i; double y; double a[11];
  for (i = 0; i <= 10; i++)
    scanf("%lf", &a[i]); /* %lf means long double*/
  for (i = 10; i >= 0; i--) {
    y = f(a[i]);
    if (y > 400)
      printf("%d TOO LARGE\n", i); /* %d means double*/
    else
      printf("%d %lf\n", i, y);
  }
}
```

2.7 Object Oriented Languages

A major application of computers is the simulation of real-world systems (for example, hospital waiting lists, industrial production lines, etc). Early programming languages were developed specifically for simulation which include GPSS and Simula 1. The designers of Simula (Dahl and Nygaard) introduced the concept of a *class* to their programs to represent simulated entities. This was the first 'object-oriented language' as we know them, and therefore Simula 67 can be considered the parent OOL. Smalltalk in 1980 and Eiffel in 1986 were influential in the further development of OOLs; which was furthered in 1985 by Bell Labs who developed C++, the OOL cousin to C. Java and C# are considered descendants of C++.

2.8 Scripting Language

When Scripting Languages were first introduced, they were brought in to automated the task which a human operator could do, for example shell scripts. However nowerdays a scripting language loosely refers to high-level general-purpose programming languages such as: Pearl, Python, PHP, Ruby, JavaScript, and Matlab. Scripting languages are generally typeless with relatively simple syntax and semantics; they are usually interpreted and are, by design, fast to learn and write in. Shown below is the TPK algorithm in Python:

```
from math import sqrt
def f(t):
  return sqrt(abs(t))+5*t**3
a = [input() for i in range(11)]
for i in range(10,-1,-1): # range() is equiv to [10,-1)
  y = f(a[i])
  if y > 400:
    print i, "TOO LARGE"
  else:
    print i, y
```

2.9 Logical Programming (Prolog)

Logical Programming is a type of programming which is built upon logical statements which are comprised of variables, constants and structures. In Prolog, variables begin with capital letters, constants are either atoms or integers and structures consist of a functor and arguments.

A Prolog program consists of facts and rules. Prolog programs are used to answer queries, although simple arithmetic operations are possible. A query is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interests.

2.10 Language Evaluation Criteria

There are a number of criteria on which the readability, writability and reliability of a programming language can be evaluated.

2.10.1 Simplicity

The simplicity of a language is defined based on the simplicity of the syntax and the number of constructs (small number of constructs is simple). Programmers will tend to only learn the part of a large language which suits them and therefore probably not use the best construct / syntax for what they are trying to achieve. It is, obviously, possible to make a language too simple for example Assembly Language.

2.10.2 Lexical Elements

The form that the individual lexical elements (i.e. words or symbols) or a language take can affect the languages' readability. The meaning of a symbol / keyword should ideally be obvious from it's name.

2.10.3 Orthogonality

Orthogonality in a programming language means that it has a relatively small number of control and data constructs which can be combined in a relatively small number of ways and every possible combination is legal and meaningful. When using an orthogonal language, a programmer is not required to remember a lot of "special cases" in the use of it's constructs. The orthogonality of a language influences both the readability and the writability of software; if a language's rules contain fewer special cases, it is easier to learn.

2.10.4 Data Types

It is also important for a language to have a rich set of data types; as well as having adequate mechanisms for combining types.

2.10.5 Expressiveness

The expressiveness of a programming language relates to how much code is required to implement computations.

Page 3

Lecture - High Level Language Implementation

2024-02-02 14:00 Jiacheng

3.1 Computers & Languages

A computer processor's circuitry provides a realisation of a set of primitive operations (or machine instructions) for arithmetic and logic operations. Some machine level instructions are called macroinstructions because they are implemented with a set of instructions at an even lower level called microinstructions. The machine language of a computer is the language which the computer understands directly, these are very simple as that is the most cost effective solution. It would *theoretically* be possible to design a processor to directly use a language that we would classify as 'high level' however this would add an extreme amount of complexity which isn't worth is.

Sitting on top of the machine language is the operating system, this is a collection of programs that supply higher-level primitives (including device and file system management, I/O operations, text/program editors, etc). Implementations of programming languages exist on top of the operating systems.

3.2 Language Implementation Methods

3.2.1 Compilation

A *Compiler* exists to translate high-level program (written in a source language) into machine code (machine language which the processor can understand). Compiling a program is slow as there is a number of checks and stages which have to be undertaken however as the output is instructions which can be directly executed on the processor, execution of the program is fast.

A Compiler is a program that translates a program in a source language into an equivalent program in a target language. The source language is a high-level language and the target language is a low-level language. Compilers are structured as an ordered series of steps which all make use of the 'symbol table' in different ways. The output from one step feeds into the input of the next step. The phases are outlined below.

3.2.1.1 Lexical Analysis

The *lexical analyser* reads the source program's text one character at a time and returns a sequence of tokens to send to the next phase. Tokens are symbolic names for the lexical elements of the source language and each token is associated with a pattern. The scanner matches patterns against sequences of input characters and when a match is found for one of the patterns, the corresponding token (and any additional required information) is output and passes to the next phase

3.2.1.2 Symbol Table

The symbol table is a data structure containing all the identifiers (together with their attributes) of a source program. For variables, the attributes could be size, type and scope; and for methods, procedures or functions - the attributes could be the number of arguments and their types and passing mechanisms and return type.

3.2.1.3 Syntax Analysis (Parsing)

The syntax analyser analyses the syntactic structure of the source program. The input to a parser is the sequence of output tokens from the lexical analyser. The Syntax Analyser applies the rules that define the language on the syntax tokens. During this process, the parser uses rules to derive the sequence of tokens. Parsers usually construct abstract syntax trees that still represent the source program's syntax, however are simpler than the corresponding parse trees.

3.2.1.4 Semantic Analysis

The syntax analyser checks whether the program is syntactically correct, but not that it is completely valid or semantically correct. The semantic analyser determines if the source is semantically valid. It uses the AST and Symbol table.

3.2.1.5 Code Optimisation

The code optimisation phase is used to shorten the time taken to run the program and improve it's space efficiency. It does this through trying to remove as much redundant code or through pre-executing code which will always return the same value in runtime, ie adding 3 and 7.

3.2.1.6 Code Generation

The last stage of compilation is to generate code for the specific machine. This phase involves: selecting which machine language instruction to use, scheduling these instructions in the most efficient order; allocating variables to processor registers and generating debug data if required. The output from this phase, and ultimately of compilation, is usually programs in machine language, or assembly language, or code for a virtual machine. The code generation can be found in compiler design texts.

3.2.2 Pure Interpretation

Pure Interpretation is a type of translation where the program we are planning on running is interpreted by another program called an interpreter. The interpreter directly executes the programs written in a high-level language, line-by-line as the program is executed. There is no pre-compilation or batch-compiling in pure interpretation.

Through Pure Interpretation - errors are more likely to occur during runtime, this is because there is no error checking as there is no pre-parsing of the code before execution. An interpreter parses the source code and executes it directly, examples include BASIC and early versions of LISP.

Pure Interpretation is much slower than compiled programs. This is because decoding high-level language statements is slower than decoded machine language instructions; which is further amplified where the high-level statement must be decoded every time the program is run.

Interpreted programs will often require more storage space to execute than a compiled program would - this is because the source code and symbol table will often both need to be present during interpretation.

Nowadays it is rare for traditional high level languages to utilise interpretation however it is making a comeback with some web languages (such as JavaScript and PHP).

3.2.3 Hybrid Implementation Systems

A Hybrid Implementation System is a mid-point between full compilation and pure implementation. A Hybrid system will compile the source code into an intermediary language which then gets interpreted at runtime through a virtual machine. The VM takes the intermediate language as it's machine language and can therefore interpret that at much greater speed.

3.2.4 Just In Time Implementation

In Just-In-Time (JIT), the source code is initially compiled to an intermediate language. The intermediate language is loaded into memory, and segments of the program are translated into machine code just before they are executed. The machine code version is kept for subsequent calls.