

Module 811

Language Development and Selection

Aim

After working through this module you should be familiar with the variety and structure of the major computer languages, and their evolution. You should also be able to place any given language in the context of other languages by its characteristics, and to define the stages involved in systematically acquiring a 'new' language.

Learning objectives

After working through this module you should be able to:

1. Understand the reasons for the proliferation of computer languages.
2. Describe the evolution of computer languages.
3. Describe the typology of computer languages.
4. Place a specific language within the language typology.
5. Select the most appropriate language for a given task.
6. Explain the circumstances under which it is necessary to learn a new language.
7. Describe the development process in any particular computer language.
8. Define the steps that can be taken to systematise the language acquisition process.

Content

Development of computer languages
Typology of languages by major characteristics
Language selection
Language acquisition

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

References & resources

Baron, N.S., *Computer Languages*, Pelican, 1986

Module 811

Objective 1 After working through this section you should be able to understand the reasons for the proliferation of computer languages.

There are thousands of human languages and dialects, although most of the world's population speak one of only three or four. This variety is a product of millions of years of evolution, including cultural isolation and contact. Yet in less than fifty years we have created hundreds of computer languages, if you take into account 'dialects'. Why some many in so short a time?

One reason for asking (and attempting to answer) this question is for the insight it might provide into the phenomenal development of the computer industry. More importantly, from our point of view, if we are able to provide some basic answers for this question we will be in a position to select the right language for a specific problem.

Language Variation

No two computer are identical, but many show similarities based on common purpose or descent. Most languages have constructs that allow logical operations, but languages created to solve problems with a specific logical base (Prolog, for instance) will have a greater range and complexity of constructs than a general-purpose language. Some languages are direct descendants of others (Algol led to Pascal which led to Modula) and therefore have similar syntax and structures. Generally, however, languages offer great diversity.

One useful way of characterising this is by examining the *purpose* of a language: what did the creators of the language expect it be used for?

Some languages — notably Basic and Pascal — were devised to teach neophyte programmers how to program, and moreover to teach good programming practice. These languages are generally versatile but lack the depth that may be required for production work, particularly when this involves system software.

Most languages are general-purpose production-oriented languages, designed for the creation of applications in business or science, or in the home market. These range from the early classics like FORTRAN and Cobol to the modern standards like C and C++. Such languages require both breadth (to support a range of application areas) and depth. They may provide complex libraries of pre-defined routines, and extensions for graphics.

Other 'languages' are really development environments, in which a whole suite of tools for application development (compilers, code generators, screen design tools) are incorporated. First exemplified by

the Smalltalk system, this approach has also been taken by building such development systems around more conventional languages such as C++.

Forces of diversity

There are a number of factors that may be important in the development of a new computer language, and that can help explain the variety of existing languages.

Creativity

The development of an entire language (even one with a restricted syntax and semantics) is a complex process, involving major creative decisions. For some computer scientists this alone may be adequate motivation.

Commercialism

Whilst the syntax of languages is not patentable, there was in the early years of the commercial computing (say 1955-1965) significant commercial advantage from having a versatile language with solid compiler support. The most notable example of this was IBM's development of FORTRAN.

Proselytism

Some languages have been created to facilitate certain programming techniques, or to advance acceptance of certain programming methodologies. Pascal was devised to embody structured techniques, and to lead programmers to a clearer understanding of the advantages of top-down system design.

Research

Some languages are designed as 'test-beds' with which to experiment. Sometimes (perhaps too often) these 'escape' into the outside world and become production languages for the creation of applications. Smalltalk might be regarded as an example of this process.

The genesis of most languages, as might be expected, is a complex combination of these (and other) factors.

Exercise 1

List some of the reasons for creating a new programming language.

Module 811

Objective 2 After working through this section you should be able to describe the evolution of computer languages.

Given the range of driving forces described in the previous section, we should not perhaps be surprised at the number of languages that have been devised. We might, however, be surprised by the ‘range’ of such languages. Prolog and Modula-2, for example, have almost nothing in common. They share a small number of English-like terms, but use completely different constructs. This is because they have a different purpose, and are not intended to be alternates: what you write in Modula-2 could perhaps be done in Prolog, but it would be impossible to create the kind of applications for which Prolog was built. On the other hand, C and Pascal are quite similar in terms of constructs and terms, but reflect their different origins and purposes in more subtle ways.

Of course, all high-level languages must eventually be reduced to executable machine code. This is the major reason that we are often able to create equivalent applications that look completely different at a source code level, but function the same when executed. It is also the reason why we are usually able to choose a language for a given project, as most hardware platforms will be able to offer us a variety of languages and development environments.

Language Evolution

As with biological evolution, it is incorrect to describe the development of computer languages as ‘progress’. Whilst languages developed in the last fifteen years are in some ways more complex and sophisticated than earlier creations, this is largely a reflection of the transfer of emphasis from engineers to programmers to users.

In the earliest years of the computer era machines were developed by engineers, and were programmed at a binary level using *machine code*. It was assumed, probably correctly, that only people with a strong background in mathematics and logic would be able to undertake this task. Also, each machine had a different machine code structure, so intimate knowledge of the machine itself was essential, and language manuals were not available.

Compilers

The development of compilers in the early 1950s was the first stage in the move towards software that could be written by non-engineers, and towards a variety of languages. A compiler is a program that can turn more abstract constructs into machine code. This made it possible to create languages that used more intelligible structures. These went in

two directions. The first level of abstraction produced *assembly languages*, where the language consists of simple mnemonics for the instruction set of the processor and a program is a set of direct manipulations of data in memory. The second level of abstraction was intended to produce programs that were intelligible at a natural-language level, using normal words (LET, ADD, REPLACE ...). More importantly, these *high-level* languages were designed specifically to allow programs to be written by codifying actual 'human' methods of problem-solving. Thus adding two numbers together might be achieved by a statement such as LET A=B+C or just a=b+c, rather than defining each memory operation required to achieve this task.

The result of this was the development of people who specialised in defining problems in such a way that they could be solved by writing a computer program, and then writing the code. Thus was born the separate job of the *computer programmer*. (More recently these design and coding tasks have often been separated, with the *systems analyst* assuming a greater rôle.) The most significant of these early *high-level languages* were FORTRAN, COBOL, BASIC and ALGOL

High-level languages

FORTRAN (*Formula Translator*) was created by a team at IBM, and its wide availability was a major component in the enormous growth of the company in the 1950s and 60s. Whilst most suited to mathematical and scientific problems, the language was (and remains) flexible enough to carry out other tasks.

COBOL (*Common Business-Oriented Language*) was designed to be a language for business applications, and reflect this in its strengths (list and file processing) and in its verbose (or non-cryptic) code. The continuing longevity is partly attributable to these strengths, but more to the inertia created by the enormous volume of existing COBOL code (and the costs involved in re-designing and re-writing in a more 'modern' language) and the number of COBOL-trained programmers.

BASIC (*Beginners All-Purpose Symbolic Instruction Code*) was the first (and most persistent) language designed specifically for teaching the programming process.

ALGOL (*Algorithmic Language*) was designed as an attempt to define a universally-applicable language, offering an alternative to the close identification between FORTRAN and IBM hardware. Whilst largely unused outside Europe, its influence was considerable, due to its being one of the precursors of Pascal. When Niklaus Wirth created Pascal he based it on his experience as one of the designers of ALGOL, but added the crucial elements of *structured programming* that he and Edsger

Dijkstra has advocated. He also intended Pascal as a teaching language, although it has been extended to become a major production language.

One of the most productive of language designers, Wirth's languages are a useful indicator of the evolution of computing in the last thirty years. Modula, the successor to Pascal, was designed to be a production version of that language, with its emphasis on structure but with a full range of libraries and extensions. This was refined throughout the 1970s and 80s, eventually becoming Modula-2. In the last decade he has focussed on the development of object-oriented systems, and on the creation of systems in which applications, the development environment, the user interface and the operating system are seamlessly linked. This is exemplified by Oberon, designed as a complete environment for a specific workstation, but portable (albeit awkwardly) to run on other operating systems. These ideas have appeared commercially so far only rarely (most notably in the NeXt machine), but the micro-kernel approach is being applied throughout the 'next generation' of micro-computer operating systems.

The 1980s have also seen the rise of so-called '4GLs' (like dBASE), languages in which applications of a specific type can be more readily developed by non-specialists, and the creation of a group of languages based on logic (notably LISP and PROLOG) and suited to problems in the fields of artificial intelligence and expert systems.

The most recent development has been the growth of software development environments that try to minimise or remove the actual writing of code, and therefore the need to 'know' a specific language. This approach includes the development of source-code generators and of application-design systems ('visual programming'), the aim being to make it possible for *users* to create applications.

Generations of languages

The most common way of describing the evolution of languages has been in terms of generations:

First Generation:	Machine Code
Second Generation:	Assembly Code
Third Generation:	High-Level Languages
Fourth Generation:	Application-Generation Languages
Fifth generation:	Logic-Oriented Languages

As with most simplifications, this system is partly helpful and partly unhelpful. As suggested before, the development of languages is driven by complex forces and the creation of new languages (and the survival of older ones) is often difficult to understand. Nor is the process in any meaningful sense a form of progress.

Exercise 2

Briefly outline the major stages in the evolution of computer languages.

Objective 3 After working through this section you should be able to describe the typology of computer languages.

There are many reasons why scientist classify things. For biologists this has always been the most important first step in the process of understanding origins and linkages. An attempt to classify computer languages has some of the same aims, but may also be useful for other reasons. Our interest in the process will largely be for the possible insights it might yield into the communality of languages, in the expectation that identifying similarities (and differences) between languages will aid in the often-difficult task of embracing new languages.

In this section we will review some of the characteristics of languages that might be used for classifying them, and we will form a basic typology based upon certain of these. A *typology* is a structure into which occurrences of objects (in this case computer languages) can be placed by defining their 'value' on a small number of criteria (usually two or at most three). In this case most of the possible criteria will be 'binary', in that we can specify two choices (high-level or low-level, compiled or interpreted), or property and its absence (structured or non-structured, extensible or non-extensible).

Simple (functional) Distinctions

There are numerous ways in which we can classify computer languages. We could use the clumsy but persistent 'generation' approach described earlier, but this would place almost all current languages into two categories: low-level (various forms of assembly language) and high-level (all other languages). We could also use the simple - but otherwise unhelpful - distinction between interpreted and compiled languages. But, as we shall see later, this distinction has as much to do with how a language is actually implemented than with any inherent feature of the language itself. Although some languages are always compiled, there are many others which can function equally effectively in either form. That C is always a compiled language is a reflection on its origins (to create system software) and its necessary employment of complex separate compiling and linking processes.

We could also characterise languages by the data with which they can be used; this is usually reflected in the variety of data types, and the emphasis. Some languages support numerical and textual data equally well, whilst others may have greater facility with numbers (eg. FORTRAN) or text (eg. LISP or SNOBOL).

Some languages are *extensible*, in that additions can be made to the basic grammar to create new grammatical elements. The dominant

imperative languages (as defined below) are not extensible in this sense; if we add extensions (as we have used the term in earlier units) to Pascal we are really creating a version (or *dialect*) of Pascal, not an extension of it in the strict sense. Languages like FORTH and Smalltalk, on the other hand are defined in a minimal manner, with the expectation that the programmer will define new elements as required by the particular application.

A more useful (and to some programmers quite significant) distinction is between those language designed to be structured, and those that either do not support structure (have no basic modular units) or have been re-designed to employ structured concepts. Again, this distinction is limited in that nearly all languages created in the last 25 years are at least partly, if not completely, influenced by the principles of structured programming developed in the late 1960s.

Conceptual Distinctions

A more sophisticated typology is possible if we employ somewhat more abstract characteristics than those defined so far. Central to these abstractions is the question of how a computer languages approaches its basic purpose, namely the creation of solutions to specified problems.

Algorithmic languages

The most common approach to problem-solving in computing is embodied in most languages: the algorithm. Problems are solved by defining a series of steps which, when carried out systematically, will result in a solution. A computer program is simply the implementation of an algorithm. The languages with which we most familiar — FORTRAN, COBOL, PASCAL, BASIC — and in which most software is written are all examples of this group of languages. They are also know as *imperative* languages.

Functional languages

An alternative approach is to treat solutions as an aggregate of applications of mathematical functions. Languages based on this somewhat abstruse idea are called *functional* or *applicative*, and are relatively little-known, if influential. The most significant has been LISP (*List Processing*); others include APL (*A Programming Language*) and SNOBOL (*String Oriented Symbolic Language*). Their major influence has been on the development of logical programming languages.

Object-oriented languages

Although we can apply OOP principles in a number of standard imperative languages, often by adding a small number of extensions to

the base language (most notably C to produce C++), there are certain languages that are object-oriented at a much more fundamental level. Rather than viewing the object as an extended data type, in languages like SIMULA and Smalltalk applications are no more (indeed, cannot and need not be more) than a set of objects that react and inter-act, using their in-built methods.

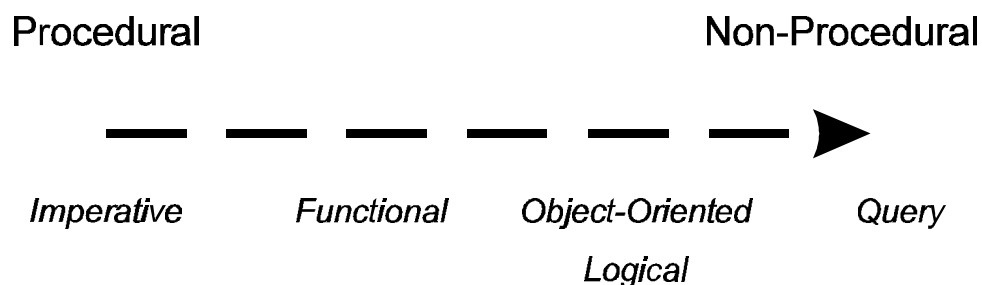
Query languages

There are a number of significant special-purpose languages, usually defined as '4GLs' that exist to allow high-level (abstract) data specification and retrieval from external structured data files (ie. data bases). Such languages include SQL (Structured Query Language) and various forms of QBE (*Query By Example*).

Procedural Distinctions

An alternative division (although not binary) is by the extent to which a language is *procedural* in form. The 'degree' to which a language is procedural is a measure of how explicitly the solution to a problem is specified in the program. Most imperative languages are strongly procedural (or *prescriptive*) of the solution via an algorithm. Once the algorithm is embodied in the program, any non-trivial change to the program represents a change to the algorithm (and hence to the solution). A full specification of the solution (the algorithm) leads to a formal description of the program; the only scope for variation is in relatively minor choices in areas such as data structures, internal modularisation, and efficiency of operation and/or coding.

The opposite approach is exemplified by query languages, in that the user (or programmer) is required only to specify the outcome (ie. the solution to be achieved), leaving the exact process (the program) to be generated by the system, using basic rules. Such languages can be defined as *non-procedural* or *declarative*. The functional languages lie more towards the imperative/procedural end of the spectrum, with the logic languages towards the declarative end. This sequence is illustrated in the figure below.



There are of course yet other ways of describing (and hence classifying) computer languages. We have used the term general-purpose, with the unwritten implication that there exist special-purpose languages. How one defines a language in these terms is especially subjective. Is COBOL special-purpose because most applications written in it are business-oriented, or general-purpose because many other types of application could be (indeed, have been) written in it? Is Prolog special-purpose because it is the pre-eminent logic programming language, or general-purpose because logic programming is applicable in a huge number of fields? Nor can we forget the process of time; languages get used for purpose for which they were not designed, or which their designers could not have foreseen.

It should clear that the classification of computer languages is a subjective process, but one not without value.

Exercise 3

Draw up a simple (2x2) table using the basic distinctions imperative vs functional and object-oriented vs not object-oriented and place the following languages in one (or more) of the four boxes: COBOL, C, PASCAL, ADA, SNOBOL, BASIC, Modula. Add any other languages with which you are familiar. Do any languages *not* fit into one of the boxes?

Module 811

Objective 4 After working through this section you should be able to select the most appropriate language for a given task.

In order to select a language for a given task we need to combine two pieces of knowledge. One is the characteristics of available languages, using the criteria and typology described in the previous section.

The analysis presented in the last section needs to be developed in a slightly different direction if we are going to use it to help us to select a language for a specific application. We must also complement the analysis of language type with consideration of what we might term extrinsic factors that might impinge upon the question of what is the 'right' language to use.

Language purpose

The first thing we need to do is to extend the discussion of special-purpose vs general purpose in a more practical and realistic direction. Allowing for the comments at the end of last section about the flexibility of languages (and the apparent perversity of some programmers), we can nonetheless make a first pass at defining which language is suited to certain types of application. Part of this is based on suitability (or potential suitability), but also on actual usage: what languages are widely used for a given task. The idea of a special-purpose language is replaced by the purposes to which a languages is put.

Of course, irrational factors are part of this system. The ability to carry out a particular task in a particular language doesn't imply that it *should* be done that way. The folk lore of programming is replete with stories of programmers writing compilers and other system software in COBOL, or a lexical parser in FORTRAN. Such stories (sometimes, but not always, apocryphal) suggest that many programmers are unable (or unwilling) to adapt their skills to the needs of the task at hand.

The other major deficiency of this approach is its emphasis on the *status quo* and the almost circular nature of the argument: COBOL is a business-oriented language (only partly true) and most commercial applications have been written in COBOL, so we define COBOL as an appropriate language for business applications, so our next business application should be written in COBOL ...

With these comments in mind, we can still define the following basic relationships:

<i>Application area</i>	<i>Typical languages</i>
Business	COBOL, Query languages

Science/Engineering	FORTRAN, Pascal
Real Time Applications	ADA, Modula-2
Embedded systems	ADA
Simulation	Smalltalk
Expert systems	LISP, PROLOG
Teaching	Pascal, BASIC, LOGO
General	C, Pascal

Many languages are genuinely multi-purpose and could appear in several categories. Others may appear equally flexible but are most effective in a particular niche (eg. ADA for real time applications).

Extrinsic factors

The selection of a language is not simply a matter of choosing the language best suited to an application without regard to the environment under which it will be developed and used. The following factors need to be taken into account.

- What language is the current system written in? Program maintenance usually requires staying with the same language (although it may not be one with which the programmer is familiar), whereas system replacement may offer the opportunity to choose a 'better' language.
- What languages are available on the expected platform?
- Are all the required features (such as file I/O, graphics, or advanced memory management) accessible in a particular language implementation/operating system combination?
- If a language is unfamiliar to the programmer or development team, is there time (and/or funding) for the necessary learning process?
- Are there 'regulatory' factors involved, such as the insistence by the US Government on the use of ADA in all software contracted for by the military?

The answers to these questions lead us to two basic conclusions. The first is that in many situations cost is a major factor in language selection, in that a better language may be feasible, but too expensive to implement. The second is that a likely outcome of a purely rational analysis of the process of language selection will often lead to the conclusion that a programmer must work with a language with which he or she has little or no previous experience. Under these circumstances, the next step is to begin the process of learning the new language.

Exercise 4

Explain in which language (or languages) you would expect (all other external factors aside) to write the following applications:

- a simple PC-based e-mail system running across a LAN
- a system for displaying machine output for monitoring patients under anaesthetic
- an accounting system for small businesses
- an interactive tutorial employing graphics and data read from a CD-ROM
- a Windows-hosted music program using the MIDI music interface to control a keyboard
- a simulation of the operation of a petrochemical works

Objective 5 After working through this section you should be able to explain the circumstances under which it is necessary to learn a new language.

Most programmers learn new languages reluctantly. The process is usually time-consuming, and the knowledge acquired (the new language's syntax and grammar) might not be used often. Of course, some programmers also learn extra languages for their own sake, perhaps to improve their employment prospects, as a form of 'one-upmanship' with their peers, or for the intellectual satisfaction.

The process is also not something that has been extensively researched. Linguistics has examined the question of how humans learn *a* language (their first), and educators have examined how we can best learn other languages. Few of the insights gained seem applicable to the different circumstances that characterise computer languages: a limited vocabulary; a syntax that is only loosely related to human languages; late acquisition within the human education process; limited opportunities for language 'practise'; and the absence of 'native

Why learn another language

Given these problems, why don't programmers simply stick with the language they first learned and use it for the rest of their careers? Apart from the motives listed above, we can identify several reasons why a new language might need to be acquired. Fundamentally, these factors underlie the selection process described in the previous section.

Availability on a particular platform

Not all languages are equally available on different systems. A programmer might move to an organisation that has different hardware, or the current organisation may change hardware. A client may already have specific hardware and be unwilling or unable to change. In each case, it is possible that the languages with which the programmer is familiar will not be available.

Specific features

Perhaps the most common reason for changing to a different language is that it provides access to features that other languages do not. To gain the advantages of object-oriented programming that were described in Modules 804 and 805 requires the use of a language that supports OOP. This precludes languages (like FORTRAN and COBOL) that are intrinsically unable to provide support for classes, encapsulation, inheritance and other key features of object-oriented programming. As most programmers have not until recently been

taught an object-oriented language as part of their training, it follows that acquiring such a language is going to be necessary at some stage. This is also true for programmers needing to work in hosted GUIs like X-Windows or the Macintosh, where development in object-oriented languages (primarily C++) is the norm.

Converting systems

Sometimes software managers are obliged to convert a system from one platform to another, usually when updated hardware is installed. This may involve only minimal conversion (re-compiling after changing library structures, for example). Less frequently, a decision may be made to undertake a total re-write of a system because the language is archaic and inadequate.

Maintenance

One of the tasks often assigned to new programmers is to take over the support and maintenance of an existing system. This system may be in a relatively archaic language, a version of the language that is no longer in common use, or simply a language with which the programmer has no previous experience.

Exercise 5

Describe a realistic scenario on your own environment in which you might be required to learn a new language.

Objective 6 After working through this section you should be able to describe the development process in any particular computer language.

The *development process* describes the stages that are required to create an application. The complexity of this process, and the stages involved, depend upon the language itself, but also upon the implementation. The process of developing a C application is different on a UNIX workstation than on a PC, even if the code is almost the same. Developing a Microsoft Windows application, on the other hand, involves much the same process in C++ or Pascal.

However, the fundamental determinants of the development process are the key characteristics of the language, most notably whether it is an interpreted or compiled language (or, if either is available, which is the relevant version).

Interpreting code

The development process for an interpreted language is simpler than that for a compiled language. All that is required is an editor to create the source code, either a stand-alone editor (if the code is in ASCII form) or within the interpreter. Programming is an iterative process whereby new code is developed in a separate test-bed program and, when thoroughly tested and debugged, transferred to the main program. The code is then submitted to the interpreter to be run.

Access to internal and external libraries is handled automatically by the interpreter, and there is often limited capacity to create or modify libraries.

It should be emphasised that this view of interpreted languages is not true for a small sub-set of those languages, most notably Smalltalk. This 'language' illustrates the increasing irrelevance of the distinction between language and development environment. It is really a complete environment in which all applications are built by using components of the system libraries, but also using code from existing applications.

Smalltalk also differs from conventional interpreted languages in several fundamental ways. It was not the first object-oriented language, but it has been the most influential and in many ways the most consistent. It is a graphical system in that the applications for which it is most likely to be used tend to involve display of images. More importantly, the development environment in all versions of Smalltalk is 'hosted' by a GUI (such as X-Windows of the Macintosh) and features an unusually wide range of visual development tools. Many of these development tools (object browsers, parsers, code generators) have since become key components of 'visual programming' systems.

Compiled code

Creating code that will be compiled to produce a stand-alone application is usually a more complex process, involving combinations of editors, compilers and linkers.

This complexity can often be hidden from the programmer. Sometimes the operation of the development environment can simplify the process to a point where it looks very similar to using an interpreter. Turbo Pascal is an example; except for generating the final executable (.EXE) file, most small or medium-sized applications can be created, tested and debugged completely inside the IDE. Only in larger applications may the IDE itself prove a hindrance, and the developer must switch to using the command-line compiler outside the IDE.

In these circumstances the programmer is exposed to a more complex version of the development process:

- using an editor (such as the one in the IDE, a general text editor, or a specialised programmers' editor) to modify the source
- submitting the code to the command-line compiler
- returning to the editor to correct errors, then re-submitting ...

In Turbo Pascal the final stage of the process (linking object files to create the executable) is handled automatically by either compiler, but in C the linker is a separate program that must be invoked manually. Whilst apparently clumsy, this separation between the compiler and linker adds to the flexibility of the C language, as it provides the basic support for the separate compilation of previously-compiled object (.OBJ) modules. As we shall see in the next module, this is one of the keys to making C a successful production system, in that it encourages the re-use of working code by allowing the generation and management of various forms of code library. It also increases the ease with which programmers can mix modules created in from source in different languages.

In its most 'extreme' form the compiler process involves up to three separate pieces of system software and a complex structure of internal and external libraries (system and user-defined). The complexity of the process has usually been assumed to be worthwhile in that it is the only way we have had to create complex systems. In recent years, however, the necessity with dealing with this level of complexity has been questioned. Even if the system needs to be complex, does the programmer have to juggle the whole system at the same time? This problem has been addressed in a whole range of environments, most significantly those involving applications for GUIs like X-Windows, OSF Motif or Microsoft Windows. Even the simplest application in this

Module 811

kind of environment needs to use a number of API calls to create a basic window/menu combination, and larger applications call upon dozens of inter-linked systems. It is in these systems that visual programming has made major strides in the last few years.

In the next module (*812: Program Development in C*) we will examine in detail how applications are developed in C and C++.

Exercise 6

Compare the basic steps in developing a program using an interpreter and a compiler.

Module 811

Objective 7 After working through this section you should be able to define the steps that can be taken to systematise the language acquisition process.

The process of how best to learn (or teach) a new computer language is not well understood, so few methods for systematising (and simplifying) the process have been defined. Also, as much of the learning process is ‘internalised’ it will never be possible to fully comprehend how this process works.

Nonetheless, it is possible for us to define a series of simple preliminary stages that anyone embarking on the task of learning a second (or later) language can work through. The final stage — of gaining familiarity with the language — will always require appropriate exercise and practice. These stages are straightforward and reasonably obvious, and are based on the foundations we have laid in previous sections.

A convenient way of approaching the task is in the form of a simple series of questions, as a sort of checklist.

1. Where does the language fit within the overall typology of languages? Is it a declarative, procedural language (like PASCAL) or a rules-based, logical language (like LISP)? Does it have support for the object-oriented paradigm, and in what form?
2. Is the language basically interpretive or compiled? If both forms exist (eg. BASIC) which does the particular implementation use?
3. What is the development process for the language? Is there an integrated development environment, stand-alone compilation/linking, or some combination of these?
4. What is the mechanism for library support, and for developing libraries?

The answers to the first two questions should allow us to gain an overall impression of the major characteristics of the language: its constructs, likely syntax, emphases etc. They will also form a basis for the critical step of comparing the new language with those languages with which the programmer is already familiar, and which can be expected to share some of these characteristics. The answers to the last two questions help to set up a smoother transition to the learning process by allowing the programmer to start the learning with some familiarity with the development environment.

An Example: A PASCAL User Learning C

We can see this process in operation by examining the decisions involved in preparing to learn C for a current user of PASCAL (some of this section foreshadows the material in the next Module).

Language Type

Both PASCAL and C are general-purpose, structured, procedural languages. Neither has 'native' support for object-oriented programming, but both support it through extensions to the basic abstract data types. As such, we should expect the following in C:

- structured constructs: sequence, looping (DO..WHILE, FOR, REPEAT.. UNTIL or the equivalent), and branching (IF, IF .. THEN, CASE or the equivalent)
- arithmetical and logical operations
- basic data types and support for extensions to include user-defined types
- adequate input/output support (keyboard, monitor, files)
- support for modularisation *via* subroutines
- object or class data type

(Subsequent comments refer specifically to Turbo/Borland Pascal and Turbo/Borland C/C++, but indicate the general level of comparison that can be made.)

Interpreter vs Compiler

Whilst interpreted versions of PASCAL do exist, most are compiled. Both Turbo Pascal and Turbo C are compiled languages that produce stand-alone executables. On the IBM-PC there are versions of both that support the development of DOS and Windows-hosted applications *via* appropriate API calls.

Development Process

Both Turbo Pascal and Turbo C feature an Integrated Development Environment (IDE) featuring an editor, a compiler and a linker. In the Turbo Pascal IDE the linker is invoked 'automatically and invisibly', but in the Turbo C IDE the linker is invoked 'automatically and visibly' (linker errors are reported). The same process applies to the command-line compiler.

Turbo Pascal allows inline compilation of object modules. Turbo C supports separate compilation of modules (which we will define in the

next module) *via* makefiles (at the command-line) and project files (in the IDE).

Library Support

System libraries are supported as units (.TPU) in Turbo Pascal and libraries (.LIB) in Turbo C. As an ANSI-compatible C, Turbo C provides all the specified, required system libraries. All are accessed by indicating in the source code which libraries are required.

User-defined libraries are supported *via* compiled units in Turbo Pascal, and external libraries in Turbo C.

Specialised support for graphics is available in both using separate hardware drivers.

The Next Step

This process cannot eliminate the most time-consuming aspect of learning a new language, the actual study and practice of the new language. With any complex language (like C) there is a great deal to learn, particularly if you need to really master the language. For some purposes, of course, less depth may be sufficient, or some topics may be put aside as irrelevant.

On the other hand, the time spent on this analysis should be worthwhile if it allows the programmer to identify the areas where the new language is most unfamiliar, and concentrate effort on those areas.

Exercise 7

In preparation for the next module, use the manuals for any implementation of C to try to make a preliminary list of the constructs defined under question one. Draw up a two-column table with Pascal in one and C in the other. For each Pascal construct (such as FOR .. DO) try to list the C equivalent. Fill in any blanks as you work through Modules 813-816.