4

4 Why have "high-level" languages?

4.1 LIMITATIONS OF ASSEMBLY LANGUAGE AND ORIGINS OF HIGH LEVEL LANGUAGES

Chapter 2 introduced to the idea of a program as a sequence of instructions in the memory of the computer.

It is possible for a programmer to write programs in this form. The programmer writes an "assembly language" program. The source text of this program is translated into binary by an assembler. The binary information representing instructions is put into the computer's memory by a loader.

Generally, programmers avoid assembly language, working instead with one of the numerous "high level languages".

Problems with assembly language programming

Assembly language programming has two main disadvantages:

- it involves far too much detail;
- it is necessarily specific to a particular make of computer.

The excessive detail comes from the requirement that the programmer choose the exact sequence of instructions and data movement operations needed for each task. Since different makes of computer have slightly different instruction sets, a program written in terms of the instruction set for one computer cannot be used on a different kind of computer.

As noted in Chapter 2, assembly language programs are built up from small fragments of code that follow standard patterns ...

- <u>Sequence pattern,</u> instructions for evaluating an expression like

  ```
  v = u + a *  t
  ```

- <u>Loop pattern</u>, dealing with a group of operations that must be executed several times

- Selection pattern, dealing with choices
- Subroutine pattern, breaking down a complex process into simpler sections.

It is easier if the programmer can use concise statements to specify the required patterns instead of having to write the corresponding instruction sequences.

e.g. this expression is easier to read than this code

```
1)
        V = U + A * T                            load   t
                                                 mpy    a
                                                 add    u
                                                 store  v

2)
        T = 0.0                                  move fzero, t
        DO 10 I=1,10                             move #1, i
        DIST = U*T + 0.5*A*T*T          l100     compare i,#10
        WRITE(5,500)T, DIST                      jgt l101
        T = T + 1.0                              fload t
   10   CONTINUE                                 30 more instructions
```

Assemblers made it possible for programmers to work in terms of readable text defining instructions rather than bit patterns. The assembler program does the translation task converting the text instructions into the bit patterns.

Compilers

High-level language translators (compilers) allow programmers to write statements. A compiler can translate a statement into an appropriate instruction sequence (which can then be input to an assembler).

Just like assemblers, compilers rely on rules that define the allowed forms for the different possible statements. The rules defining the form of an assembly language are very simple:

```
•   a line of an assembly language program can be
           an optional label and an instruction
    or
           a variable name and an initial data value

•   an instruction can be
           an i/o instruction
    or
           a data manipulation instruction in the form

                  instruction-name operand
    or
    ...
```

Translation involved little more than looking up the instruction name in a table to get the corresponding bit pattern.

Syntax rules for high level languages

The rules defining the form of a high level language are naturally a lot more complex than those for an assembler because we want a high level language to allow the direct use of more complicated constructs. These rules defining the allowed forms in a language define the *syntax* of that language.

4.2 HIGH LEVEL LANGUAGES CONSTRUCTS

4.2.1 Statements

Statement types

A language will typically allow for a number of different kinds of statement:

- assignment statements (define a new value for a variable in terms of some expression that combines values of other variables, e.g. `V=U+A*T`)
- selection statements (provide something that allows the programmer to indicate selection of one operation in a set of two or more alternatives)
- iteration statements (loop constructs)
- subroutine call statements

  ….

Most high-level languages still exhibit relics of their assembly language ancestors. A program is still a sequence of statements that are normally executed in sequential order (with appropriate adjustments for subroutines, loops and selection statements). Each statement may expand to many instructions, but it is still the same "instructions in the machine" model of computations.

The people who invented a particular high-level language will have chosen the forms of statements allowed in that language. The chosen forms will represent a trade off among various factors. They have to allow programmers to express complex constructs succinctly (and therefore they may be moderately complicated) but at the same time they have to be simple for an automatic translation program to recognize and interpret.

FORTRAN

The first few examples in this section use FORTRAN statements. FORTRAN is about the oldest of the high level languages (first used in 1956) but it is still quite widely used (the language has been updated a few times).

The forms of statements in the earliest dialects of FORTRAN are obviously related to simple, standard patterns of machine instructions as would have been familiar to assembly language programmers in the 1950s. (Some critics of the continued use of FORTRAN point to the absurdity of still writing code for the assembler used on an IBM704.) For example, the early FORTRAN dialect provided programmers with a simple "arithmetic if" test as one of its few conditional constructs ...

```
C      CHANGE MAXTEMP IF TEMP GREATER THAN CURRENT MAXTEMP
       DIFF = MAXTEMP - TEMP
       IF(DIFF) 150, 160, 160
150    MAXTEMP = TEMP
160    CONTINUE
```

In FORTRAN, lines starting with a 'C' were comment lines that a programmer could use to explain what was being done in the code. The 'assignment statement' on the second line (DIFF = …) calculates the difference between the current maximum temperature value and some newly calculated value. DIFF will be positive if the current value of MAXTEMP is already larger than new value. This code uses an 'arithmetic if' statement. The part IF(<variable name>), like IF(DIFF),had to be translated into a sequence of instructions that would test whether the value of the variable was negative, equal to zero, or positive. The rest of the 'arithmetic if' statement comprises three label numbers. These label numbers identify the statement that should be executed if the value was negative, 0, or positive.

In the example program, if DIFF is negative, the MAXTEMP value must be changed. This change is done by the code (starting at) statement 150. If DIFF is zero or positive, then MAXTEMP is equal or greater than the new value, and so no change is needed. Rather than execute statement labelled 150, the CPU executing the program can jump to statement 160.

The simple structure of the "arithmetic if" statement made it easy to define

1) rules that a compiler could use to recognize such a statement

and

2) a small code template that the compiler could use to generate code, e.g.

```
    IF(<variable>) <label1>, <label2>, <label3>

translate to
    fload         <variable>    (loads real (float) number
                                into a CPU register)
    ftest                       (get "flags" set)
    blt    <label1>             (if "less"  flag set ,
                                       branch to label1)
    beq    <label2>             (if "equal"  flag set ,
                                       branch to label2)
    jmp    <label3>             (otherwise go to label3)
```

The arithmetic if statement was one of the simplest in the FORTRAN language. Other constructs, like DO ... (the construct used to form iterative loops), required much more elaborate code templates for their translation (as well as more complicated recognition rules). The template for translating a DO ... construct would have needed to use something like 20 assembly language statements specifying the various instructions needed. These would have come in two groups; one group at the start of the iterative code, the other group at the end.

These high level IF()... and DO ... constructs saved the programmers from having to sort out all the specific instructions required, so giving them more time to think about the best way of coding a problem.

Productivity gains

Numerous empirical studies have shown that programmers generate about the same number of lines of code per day irrespective of the language that they use. It is a surprisingly low number of lines (lets say '50' to protect the programmers' dignity). If you program in assembly language, you get one instruction per line so you are getting about 50 instructions per day. If you program in FORTRAN (or another higher level language), your one line of code would generally translate into five or more instructions, so 50 lines of code meant that you were coding at rate exceeding 250 instructions per day. High level languages give a big boost to programmer productivity.

Machine independence

In addition to getting improved programmer productivity, high-level languages give you machine independence. There might have been other computers around that didn't have a "test" instruction, nor instructions like "branch if less flag set" (in fact, the other CPU might not have had any status flags). A programmer using a high level language like FORTRAN didn't have to bother about such differences; the compilers took care of such details.

The "arithmetic if" statement illustrated earlier would have had to have a quite different translation into machine instructions ...

```
IF(<variable>) <label1>, <label2>, <label3>
```

might, on the second machine, be translated using the template

```
fload      <variable>    (loads real (float) number
                         into the CPU's float register)
spfa                     (skip, i.e. miss out next
                                instruction, if the float
                                register contains +ve value)
jmp        <label1>      (deal with -ve case)
snfa                     (skip if register value non zero)
jmp        <label2>      (deal with zero case)
jmp        <lable3>      (deal with +ve case)
```

If you had equivalent FORTRAN compilers on both computers, you weren't bothered by such differences. You could rely on the compiler for a particular machine to use a template with the appropriate set of assembly language statements that used the correct instructions for that machine.

4.2.2 Data types and structures

A high-level language and its compiler do more than make it easier for the programmer to code the data manipulation steps.

High-level languages allow a programmer to declare restrictions on how particular data elements are to be used. After a compiler has read such declarations, it will remember them and when it sees reference to variables later in the code it will check whether the restrictions are being obeyed. Such checks cannot be done in assembly language.

When writing in assembly language, a program can only define a variable as "something" that occupied a particular location in memory. If part of the program treats that variable as if it held a real number, and another part treats the variable as an integer number, the assembler program won't detect any errors. It will produce executable binary code which will give results that are almost certain to be incorrect.

High level languages mostly use explicit variable declarations that define the "type" of a variable (sometimes, a language may rely on a naming scheme where the type of a variable can be inferred from its name). Thus, in some versions of FORTRAN one could have declarations for variables like the following ...

```
REAL MAXTEMP, MINTEMP
INTEGER TIME
...
```

Such declarations were usually put at the start of a program (or subroutine) or sometimes in the code at a point just before the place where a variable was first used. Such declarations allowed the compiler to check that, for example, `MAXTEMP` was always treated as a real number.

The original 1956 FORTRAN compiler also introduced mechanisms that allowed a programmer to define "structure" in their data. "Data structures" allow a programmer to express the fact that their may be some conceptual relationship between different data variables.

Ideas of data structures have been developed substantially over the years, the original FORTRAN idea was quite limited; it was simply a way of allowing the programmer to express ideas about "arrays" (or "matrices"). Taking again the example of an engineer writing a program to model heat flow along a metal beam. It would be necessary to calculate temperatures at several points along the beam and across the beam. For example a small study might need ten divisions along the beam, with four positions across the beam at each division (Figure 4.1).

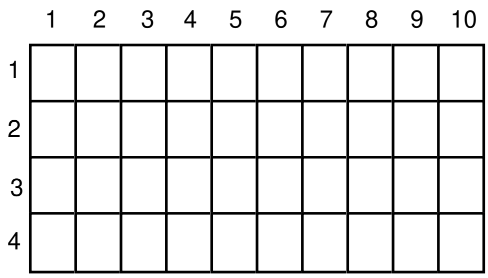

Figure 4.1 An array, or matrix, of places where an engineer wants to analyze properties of a steel beam that gets represented as an array of data values in a program.

The program would have to calculate forty different values; but the engineer programmer really wouldn't want to think of these as being 40 different variables (`BEAM1`, `BEAM2`, ..., `BEAM40`) because that would hides the nature of the physical system that the program is supposed to model.

A programmer working in assembly language would not be able to specify anything clearer than a need for space in memory to store 40 numbers. A programmer using FORTRAN could specify an "array structure" that helped make it clearer that these 40 numbers really did represent something involving 4 "rows" of ten values each ...

```
DIMENSION  BEAM(4,10)
```

As well as helping to clarify the meaning of the data, such a declaration of an "array structure" allowed simplifications of code. Rather than a whole set of different formulae to calculate the value for BEAM1, BEAM2 etc, the program could be written using nested loops and a single formula that calculated the value for BEAM(I,J) (with I and J being variables that ran through the range of row and column positions)...

```
      DO   10  I = 1, 4
      DO    9  J = 1, 10
      ...
      BEAM(I,J) = ....
 9    CONTINUE
 10   CONTINUE
```

4.3 EVOLUTION OF HIGH LEVEL LANGUAGES

The original FORTRAN compiler of 1956 started a massive amount of development work on "programming languages". That early FORTRAN was a vast improvement over assembly language, but was also limited and flawed. Subsequent languages have tried to remove limitations, add expressive power, and employ variations on the basic "sequence", "iteration", "selection", and "subroutine call" patterns that provide fewer opportunities for errors.

Reducing opportunity for errors:

That FORTRAN arithmetic if statement is confusing; e.g. :

```
IF(NUM) 150, 160, 170
```

If NUM is positive we go to statement 150, or would it be statement 170?

As the different pieces of code are scattered around in the text of the program, and reached by jumps backwards and forwards, it is very difficult for someone to read the code and get a clear idea of what is supposed to be happening.

The complete FORTRAN example given earlier was supposed to replace MAXTEMP with some new value TEMP if this was larger. The FORTRAN was

```
       DIFF = MAXTEMP - TEMP
       IF(DIFF) 150, 160, 160
 150   MAXTEMP = TEMP
 160   CONTINUE
```

More modern languages (Pascal, C etc) allow a much clearer expression of the same idea: e.g. in Pascal one can have something like ...

```
if(temp>maxtemp) then maxtemp := temp;
```

This clearer style makes it much less likely that a programmer will make a mistake when coding.

Greater expressive power:

The original FORTRAN only had "counting loops" e.g.:

```
      DO 100 I= 1, 25
      ...  (body of loop doing some work)
100   CONTINUE
```

This is fine if you know that a loop has to be executed 25 times (with the variable I counting from 1 to 25).

But often there are other conditions on which you want to iterate, e.g.:

- loop reading commands until user types 'quit';
- repeatedly process successive data elements until a negative value is found;
- ...

Later languages have their own variations on the counting loop construct; in most languages this will be a "for" loop, e.g. Pascal's

```
for i:=1 to 25 do begin
   ...
  end;
```

But the more modern languages will have other constructs such as "while" loops and, maybe, "repeat" loops; e.g. a Pascal "while" loop to process non-negative input values:

```
readln(val); (* read first value *)
while(val>0.0) do begin
    ...         (* code to process value *)
   readln(val); (* get user to input next value *)
   end;
```

More data structures and more compile type checking

FORTRAN is pretty much limited to arrays of simple real (or integer) numbers. Modern languages provide other built in types (e.g. character data for text processing programs, "boolean" data – true/false values) and give the programmer many more ways of building elaborate data structures from simple built in data types.

A large part of the second and subsequent programming subjects in most computing science courses are largely devoted to exploring the definition and use of data structures.

The compilers (particularly those for advanced languages like C++) do lots of checking to make certain that any structures defined by a programmer are used only in appropriate ways.

4.4 FORTRAN

FORTRAN was the first of the high level languages. In 1956, IBM started supplying a FORTRAN compiler to customers of its 704/709 series of computers (these were machines intended for scientists and engineers).

The name FORTRAN came from "FORMula TRANslator" --- that is how the developers thought of their compiler, it "translated" scientific and engineering formulae into assembly language code that specified the appropriate sequence of instructions. IBM didn't claim any exclusive right to the FORTRAN language and fairly soon other computer manufacturers made their own FORTRAN compilers (along with compilers for similar, but less successful languages as invented by companies or universities).

Standardization

Since one of the reasons for using a high level language was machine independence, it was recognized that different FORTRAN compilers would have to be standardized so that all used exactly the same kinds of statements and data declarations. Standardization initially involved semi-formal agreements among the computer companies but later the US government's "American Standards Association" became involved. By the late 1960s, there was a government approved standard FORTRAN dialect (ASA FORTRAN IV).

Standard FORTRAN was subsequently extended to incorporate ideas invented for other languages.

Program organization

FORTRAN defined a program as consisting of a "main program" along with any number of subroutine units.

Subroutines were divided into "functions" and standard "subroutines". Functions were short routines that simply calculated a value e.g. finding the sine() or cosine() of some angle given in radians.

Independent compilation of subroutines

Some compilers could handle a file (actually a deck of cards in those days) that contained more than one routine; but really each separate routine was thought of as a separately compilable unit. This support for separate compilation encouraged the development of libraries of FORTRAN subroutines that had been written to handle subtasks that occurred frequently in engineering applications

Linking loader

A "linking loader" was used to put together a FORTRAN program from its separately compiled parts and library routines, see Figure 4.2.

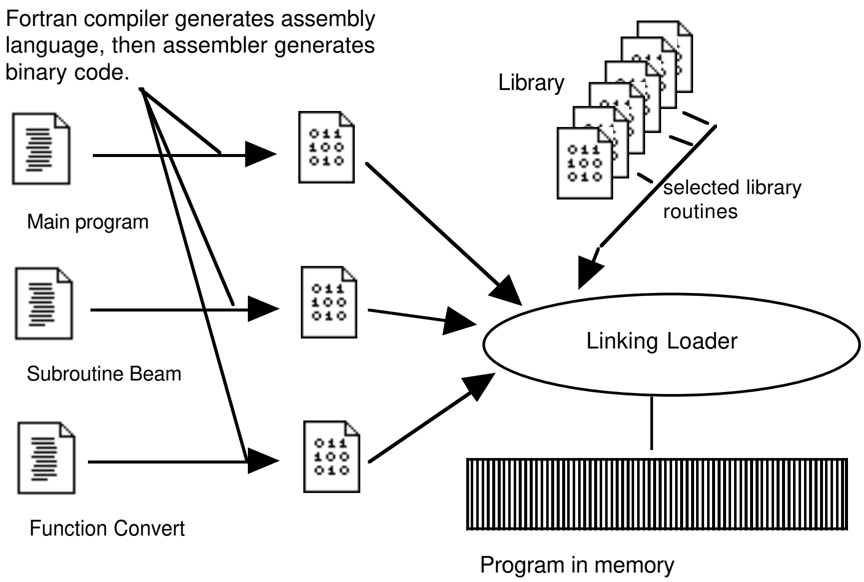

Figure 4.2 Linking a program from separately compiled and library files.

All the data variables used by a program were defined by declarations in the main program, its functions, and its subroutines. The compiler and link-loader worked together to choose fixed memory locations that would be used for these variables. In early systems, the space reserved for variables was usually located just after the code of a unit (see Figure 4.3). This "static" allocation scheme is a bit restrictive and later languages offer alternative, more flexible but more complex schemes.

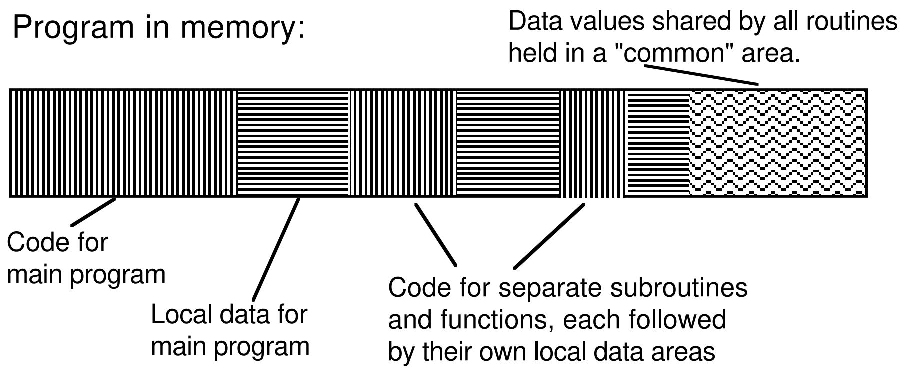

Figure 4.3 Simple arrangement of code and data areas in memory (as used by early FORTRAN compilers and similar systems).

Although generally a good way of putting together a program out of parts, the original FORTRAN scheme didn't allow for much checking for correctness. For example, a subroutine or function might need to be given a real number as input (e.g. `FUNCTION SINE(X)` wants a real value for `X`) but by mistake the programmer writing the main program might have called the function asking it to work with an integer value. Such an error would pass undetected, the program would be built and would run but would give incorrect results. Later languages have explored alternative ways of putting programs together, ways that do allow checks for consistency.

Data declarations

Each individual unit (main program, subroutine, or function) would consist of data declarations and code statements. The data declarations named individual data elements

Simple variables

```
REAL MAXTEMP
INTEGER TIME
```

and structured composites of simple data elements (the arrays):

Arrays

```
DIMENSION BEAM(4,10)
```

Statements

The statements included:

- assignments:

```
DIST = U*T + 0.5*A*T*T
```

- conditionals:

```
IF(MAX) 500, 510, 520
```

(there were several other forms of IF statement apart from this arithmetic IF),

- iterative loops:

```
      DO 100 I=1,50
      ...
100   CONTINUE
```

and

- calls to subroutines:

```
CALL MODELHEATFLOW
```

Arguments for routines:

Main programs and subroutines have to communicate.

A subroutine will have been written to take some data values as input and perform various calculations that change those data. But which data elements does a routine work with?

Sometimes, a subroutine will only work on one set of data. In this case these data elements can be declared by the main program as being in some "common" area available to all routines.

But often you want a routine (or function) to work on different data elements, e.g. you might want to calculate the sines of several different angles (represented by several different data elements in the main program). In these situations, the calling program must have some way of telling the subroutine about the specific data element(s) it is to work with.

"Passing arguments to a subroutine"

There are many ways that a calling program can tell a subroutine what data to work with (you will be meeting some of these alternative ways when you start programming). The scheme used in FORTRAN relies on the main program telling the subroutine where in memory the data can be found (it does this by "passing" the addresses of the arguments to the subroutine or function).

For example, a subroutine for sorting elements in a vector with up to 100 elements would need to be told which array to sort and how many elements it actually contained. Such a subroutine would specify this in its header:

```
SUBROUTINE SORT(DATA,N)
DIMENSION DATA(100)
...
END
```

A main program might contain more than one call to this sort routine, with different arguments in each call, so that it could get the contents of more than vector sorted ...

```
DIMENSION HEIGHT(100), RANGE(100)
...
...
CALL  SORT(HEIGHT,NCOUNT)        pass address of
                                 vector HEIGHT and
                                 variable NCOUNT to
                                 sort subset of data
                                 in vector
CALL  SORT(RANGE, 60)            pass address of
                                 vector RANGE, and
                                 address of a compiler
                                 invented variable set
                                 to contain 60.
```

I/O package

FORTRAN came with a fairly good library of input and output routines. These allowed programs to save data to tape (and disk), to read previously saved data from tape, to read "cards" with data, and to print results on a line printer.

"Formatting" of output

The routines for reading "cards" and printing provided for elaborate "formatting" of data (i.e. there was a way of specifying the layout, or "format", of data on a printed page).

Use of language

For something that started as a little project to help engineers write assembly languages on an early computer, FORTRAN has had an enormous success.

For those who have to code up engineering calculations, FORTRAN provides just about everything needed and it is not hard for compilers to generate good assembly language code. Consequently, the language is still widely used by engineers. The CPUs for the "supercomputers" used for large engineering calculations have often been designed with FORTRAN in mind. (Modified dialects of FORTRAN are sometimes the only programming language for special parallel supercomputers).

But for anything other than engineering calculations, FORTRAN is clumsy and most programmers prefer to work with other more sophisticated languages.

4.5 BASIC

Basic was originally developed at Dartmouth University in the USA in the early 1960s.

It was intended to be a simplified version of FORTRAN-II (a revised FORTRAN dialect of the early 1960s) that could be used for small programs entered and run interactively on an early time-shared computer system. (It simplified FORTRAN in a number of ways such as by ignoring the differences between real and integer values, and by restricting the number of data variables you could have.)

Interpreted languages

It was also an interpreted language. BASIC code was not translated into assembly language and thence to loadable binary. The BASIC interpreter would start by simply checking that it could recognize each statement in the program. (The statements were similar in type to those of FORTRAN with assignments, `IF` tests, loops, and "`GOSUB`" calls.) The program would then be run with the interpreter stepping through the successive statements.

As each statement was reached, the interpreter would reanalyze it and then immediately carry out the operations required. So, for a statement like:

```
110   LET V = U + A * T
```

the interpreter would get the value for variable `U`, save the value temporarily, get the value of `A`, multiply by the value of `T`, add the saved temporary value and store the result in `V`.

Costs of an interpretive system

An interpretive system is far less efficient than a compiled system. A BASIC interpreter might have to execute tens of thousands of instructions to evaluate something like `V=U+A*T` (whereas only about four or five instructions of compiled code are required). So one would never use an interpretive approach for large scale numeric calculations.

Advantages of an interpretive system

But an interpreter has some advantages. Interpreters avoid the complexity of the separate compilation, assembly, and link-loading steps that are needed by a compiled language. Interpreters can often detect "syntax errors" (where a statement is not recognizable) immediately after the user types in a line of the program; the errors can be identified and the user gets a chance to correct them. Often, interpreters can provide extensive run-time checks so if something goes wrong while a program is executing, it doesn't just stop, instead, the interpreter generates some explanation as to what is likely to be wrong. Again, this makes it simpler for the programmer to find and correct errors.

BASIC popular as an introductory language

Such advantages seemed to make BASIC an attractive environment for students and school children to learn how to program. For a long time (from 1962 until the mid 1980s) BASIC was used in introductory courses but it gradually fell from favour. The structure of BASIC was based on FORTRAN, and it shared FORTRAN's faults of error prone constructs, and a style of code with too many jumps back and forth between labelled statements. Programs written in this style are difficult to get to work correctly, and even more difficult to maintain.

Most programmers prefer working with programming languages (mainly from the Algol family) that have more structure in their basic loop constructs, conditionals etc. (Modern dialects of BASIC have adopted some of these structured programming constructs.)

4.6 LISP

Lisp is another very different example of an interpretive language. Lisp first appeared in 1960, a product of research on "Artificial Intelligence" being carried out at Massachussetts Institute of Technology.

A language for symbolic computations

Lisp is not a language for writing programs that do calculations --- it is intended for "symbolic computation". What is "symbolic computation"?

A numerical computation:

Calculate the value of the polynomial when x = 3.8

$$5x^4 - 7x^3 + 4x^2 + 2x - 11$$

A symbolic computation:

Differentiate the polynomial

$$5x^4 - 7x^3 + 4x^2 + 2x - 11$$

In the case of numeric computation, the programmer has simply to code up the formula so that the computer can work out the answer:

```
X = 3.8
VAL = (((5*x - 7)*x + 4)*x + 2)*x -11
```

For a symbolic computation, the programmer must create a much more complicated program that can read in some representation of the polynomial and produce output that represents the polynomial's derivative:

$$20x^3 - 21x^2 + 8x + 2$$

(Most students will get to use a program called Mathematica that does this kind of symbolic calculation. Mathematica is a descendant of a large Lisp program called Mathlab; the current version of Mathematica may have been written in C rather than Lisp.)

Lisp was invented to handle all kinds of non-numeric calculations. As well as "mathematical applications" like symbolic differentiation and integration, Lisp has been used to write programs that:

- translate English to Spanish or to other natural languages,
- "understand" children's stories,
- interpret photos of rooms and pick out desks, chairs, doors etc,
- execute rules that diagnose microbial infections
- check architectural drawings to make certain they comply with local government ordinances
- solve those "analogy" problems you get in IQ tests
- infer rules that capture regularities in set of data

Dynamic data structures

It is very difficult to determine in advance what data elements are going to be needed in the kind of application where Lisp is used. Examples:

- the number of terms needed varies with the polynomial given as input
- the number of chairs, doors, desks etc varies with the picture used as input
- the number of inferences made while determining the cause of a patient's infection depends on both the data that define the patient's clinical history and on the organism causing the infection (some are easy to identify and don't involve many rules, others require lots of rules to discriminate among less common types of organism)

Free storage

Instead of having all the data variables allocated before the program started, a Lisp program started with a large collection of "free storage". This "free storage" is allocated dynamically --- pieces being used as and when a program needs them. Whenever a Lisp program needs a new data element, it takes pieces of the free space and uses them to build up the data structure required.

Other languages subsequently copied Lisp and provided their own forms of dynamic storage (though most copies are actually less cleverly implemented than the original scheme used in Lisp).

4.7 COBOL

COBOL is yet another language from around 1960. Its name stands for "COmmon Business Oriented Language" and it is intended primarily for applications like record keeping and accounting.

Businesses had been using files of record cards to keep track of customers since the late 1890s. These record cards were processed using electromechanical sorting machines, tabulators (which printed data in tabular form on sheets of paper), and more complex accumulators (that added up numeric values recorded on the punched cards). Computers started to be used for such record keeping applications and accounting from about 1949. The general public probably first heard of computers when a Univac computer appeared on television in 1952 while being used to predict the result of the US presidential election.

Initially, commercial applications programs had to be written in assembly language. Of course, assembly language was just as unsuitable for commercial applications as it was for engineering calculations. So high level languages started to be invented for commercial work.

DOD requires a standard data processing language

In the late 1950s, each computer manufacturer was developing their own commercial/business data processing language. The US's Department of Defence was buying computers from many manufacturers. It needed "business" programs (e.g. programs to do the army's payroll, programs to keep track of munitions in warehouses, etc). It wanted its programmers to be able to write programs that could run on any of its computers. Consequently, it was not pleased to find that the languages were different! The DOD sponsored a consortium of computer manufacturers to come up with a common language —— COBOL.

On the whole, COBOL is not an interesting or attractive language. It is clumsy. It is verbose. It has a poor subroutine structure. But it did do one thing very well: it allowed programmers to define "data records" and provided support for files of records on tapes and disks.

Business data processing and "records"

Typical business processing applications need to keep track of things like "customers", or "employees", or "hospital patients" etc. Each such thing is characterized by many separate data values, e.g. for a hospital patient one might need:

> patient's name, patient's given names, age, sex,
> insurance number, account number,
> date of admission, ward number, bed number,
> doctor in charge, reason for admission,
> info on surgery required
> bill
> ...

These many different values all are parts of the same thing.

If these data were represented by separate variables in a program, one would lose track of the fact that they belonged together. COBOL (and some of its defunct predecessors) allowed programmers to define "records" (they were inspired by "record cards" that businesses had previously used to keep track of things like customers).

Record data structures

A programmer could declare a record, naming and defining the types of constituent fields. So a "patient record" might be defined as having a 'name' field (with space for 30 characters), a 'given names' field (space for 60 characters), account number (an integer with 6 digits), a ward number (integer with 2 digits), and so forth.

Given a declaration of such a patient record, a COBOL compiler would then allow the programmer to have variables of type "`PATIENT`". The compiler would arrange for the variable to be represented by a block of memory sufficiently large to hold all the constituent data fields of a patient. The language then allowed the programmer to refer to data fields within a particular `PATIENT` record (this helped make it clear that related data fields did belong together).

Record files

As well as having `PATIENT` records in memory, a COBOL programmer could specify that the program needed to use a file of such records held on tape or disk. The compiler would provide lots of support for operations like reading a complete record from file into memory and writing updated records back to files.

The idea of a "record" grouping together related data is very useful in most applications, not just business data processing. Modern languages like Pascal, C, and C++ incorporate the idea of records. You will start to use records (simple things like "patient" records) towards the end of your first programming course and spend considerable time on more elaborate record structures in many of your later courses.

4.8 ALGOL

1960 again! A good year for programming languages, Lisp, COBOL and *Algol-60*.

FORTRAN had just happened. No-one designed it. Its developers had simply sat down to write a "Formula Translator" that would simplify the work of those writing programs to do scientific or engineering calculations. It had no style, no elegance.

A number of mathematicians wanted something better: a language for expressing algorithms, all kinds of algorithms including recursive ones. After an abortive effort in 1958, in 1960 a group of mathematicians came up with a design for an ALGorithmic Language.

A mathematicians toy?

Algol really was conceived of mainly as a mathematical device. It was meant to allow mathematicians to express complex algorithms: algorithms for mathematical tasks like "inverting matrices", finding "eigenvalues of matrices", solving linear equations, calculating recursive functions. It wasn't really thought of as a practical programming language, for example, the original version didn't bother to define any mechanism for input of data or output of results. But it was mathematically elegant.

Algol was too elegant to be used simply as a language appearing in mathematics journals. So, people started to try to implement it as a programming language. A few of these implementations were successful and Algol was used a bit as a programming language in the '60s. Although not wildly successful as a practical language, Algol did serve as a starting point for most subsequent developments of new programming languages.

Stack based function calls and recursion

Algol couldn't use the simple mechanisms that FORTRAN could employ to organize data storage and function calls. Instead, it had to have a more complex mechanism based on a "stack" storage structure. This was because Algol had to handle "recursive" functions; the mathematicians insisted on having these. At the time, this need for a stack all seemed obscure, unnecessarily complex and difficult; but now it is the standard. All modern languages use this approach rather than the older simpler static scheme of FORTRAN.

Recursive functions? !

You are unlikely to have had to deal with many recursive functions in your earlier studies of maths. (quite possibly, you'll not have encountered any recursive functions). There are relatively few simple examples of recursive functions (and most of these are unrepresentative because they relate to problems where there are alternative iterative solutions). Real recursive functions turn up in obscure areas of mathematics and, somewhat oddly, in the context of tasks like finding a route through a maze or a way around a network of nodes and edges. Many of the more elaborate data structures now used in computing actually represent things like networks, and so you will be meeting some more realistic recursive functions when you start to write programs that use such structures.

A problem using recursion

The following is a slightly contrived example of a problem that can be solved using a recursive function.

You have to print the value of a positive integer that represents a result from your program. Unfortunately you are using a language whose inventor failed to provide any output mechanism other than the function "putchar(<character>)" which prints a character (e.g. the character 'A', or the digit character '3' or whatever else you want). So, you are going to have to find a way of generating the digit characters that represent your number and getting these printed using the provided `putchar()` routine. For example, if your number is three hundred and twelve, you'll need to call `putchar()` to print a digit '3', then another call to print digit '1', and finally a call to get digit 2.

How could you get the digits?

Getting the low order digit is not hard. The circuits in the CPU that do division can also give you the remainder value from a division:

```
e.g.     78   divided by ten   leaves remainder 8
        312   divided by ten   leaves remainder 2
```

Programming languages will have "operators" that allow you to ask for this remainder. e.g. in C (and C++) you could have

```
remainder = number % 10;
```

Doing the remainder (`%`) operation gives you a numeric value in the range 0..9. You still have to get a printable digit character --- '0', '1', '2', ..., '8', '9'. There would be many ways of doing this (and they aren't really important in this example). One way might be to use your remainder value to select the right digit from an array of characters (`char` is C's abbreviation for character)...

```
char  digits[] = { '0', '1', '2', '3', '4', '5', '6',
                      '7', '8', '9' };
 ...
remainder = number % 10;
char  correctdigit = digits[remainder];
putchar(correctdigit);
```

So, you know that it is easy to print the last digit of the number.

You could deal with printing a complete number by getting someone else to print all the leading digits, then you could print the last digit.

What would the leading digits be? Well, they will be the digits that represent your number divided by ten .e.g. you get given the value seven thousand, eight hundred and twenty four to print, you can do a remainder and see that the last digit should be a '4' but you'll need someone to print the rest – the digits that represent the number seven hundred and eight two.

That solves it! We can get the problem solved by a employing bureaucracy of people each of whom follows the same data processing rules, see Figure 4.4.

In an Algol-based language you can define recursive routine that simulate the workings of such a bureaucracy (the following is simply an Algol-ish pseudocode, the statements don't exactly match any real language):

```
recursive routine PrintPosNumber(integer Number)
begin
   if(Number<10) then PrintDigit(Number);
   else begin
         integer quotient;
         integer remainder;
         quotient = Number / 10;
         remainder = Number % 10;
         PrintPosNumber(quotient);
         PrintDigit(remainder);
         end;
end;
```

A routine has a name, e.g. `PrintPosNumber` (for print positive number), and an "argument list" (a specification of the data it needs to be given).

```
recursive routine PrintPosNumber(integer Number)
```

Here, only one integer data value has to be given to the function.

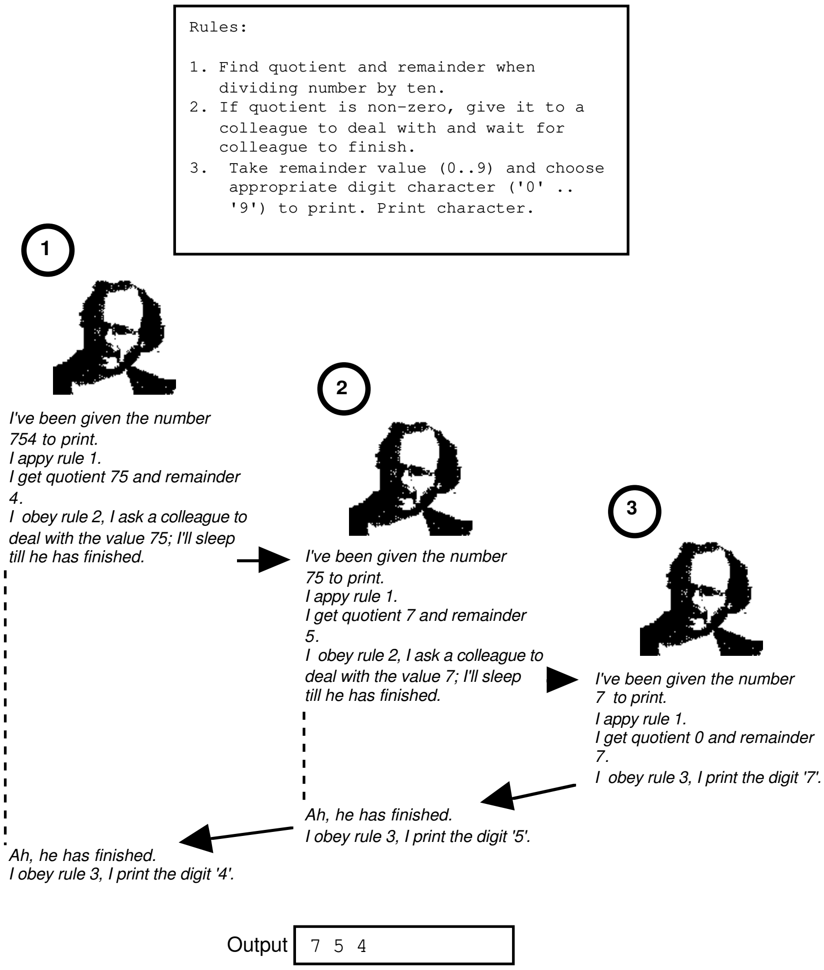

Figure 4.4 A "recursive" problem solving process.

The code checks for the easy case first (always a good idea for recursive routines):

```
begin
   if(Number<10) then PrintDigit(Number);
```

If the number is less than ten, it can be converted directly to a digit character and this character can be output using the given `putchar()` routine. This would be done by an auxiliary routine, "PrintDigit." (Usually, it is better to have lots of little routines rather than one complex routine that does all the work.)

If the number exceeds ten, it has to be broken down to get a quotient and a remainder.

```
integer quotient;
integer remainder;
```

```
quotient = Number / 10;
remainder = Number % 10;
```

The quotient has to be passed in a recursive call for its digits must be printed first.

```
PrintPosNumber(quotient);
```

Once the quotient has been dealt with, this routine can print the digit corresponding to the remainder.

```
 PrintDigit(remainder);
end;
```

At the point where the first character is printed, there will be many versions of PrintPosNumber running. In fact, there will be one PrintPosNumber for every digit in the decimal representation of a number. A five digit number, like sixteen thousand one hundred and nine will need five PrintPosNumbers working; an eleven digit number would need eleven versions. Each version has to keep track of its own "Number", "Quotient" and "Remainder" and needs some place in memory to store these values. This is where a "*stack*" gets involved.

Stack

A compiler for Algol (or any similar language) generates code for (recursive) routines that allows them to claim space from a "stack" as they start and then to release this space when they finish. The stack itself is simply a big chunk of a program's main memory that has been reserved for this purpose.

Stack frames for arguments and local variables

Routines reserve space for their input arguments ("Number" in the example) and local variables ("Quotient" and "Remainder"). A little more space is needed for 'housekeeping' purposes (like remembering a return address when another routine has to be called). The stack space reserved by a routine is called a "stack frame". Figure 4.5 illustrates the frames on the stack at the moment where the first digit is about to be printed (same example data as Figure 4.4).

Stacks have advantages over "static" allocation

Although invented for the somewhat atypical case of recursive routines, the stack based scheme for allocating memory proved to be much more useful than expected.

FORTRAN's static allocation scheme for arranging space for variables tends to waste space. Often a routine will need lots of local variables, but these only really need to have space allocated while that routine is being executed. If space is allocated statically as in a simple FORTRAN system, then its reserved for the entire time that the program is running. If all routines were to get their space allocated automatically on the stack when they started, and they freed the stack space automatically as they finished, then the space need for a routine's local variables would only be reserved while the routine was executing.

"Automatic" storage

Most current languages copy Algol and make use of a stack for organizing the memory needs of routines. The stack is also used for the housekeeping work of keeping track of the sequence of subroutine calls, return addresses etc. This "system's stack" is all handled by runtime support routines and extra bits of code inserted by the compiler. Programmers don't really notice its being there, it is all automatic. (In fact, stack based storage is often called "automatic storage".)

```
recursive  routine  PrintPosNumber(integer  Number)
begin
   if(Number<10)  then  PrintDigit(Number);
   else  begin
          integer quotient;
          integer remainder;
          quotient = Number / 10;
          remainder = Number % 10;
          PrintPosNumber(quotient);
          PrintDigit(remainder);
         end;
end;
```

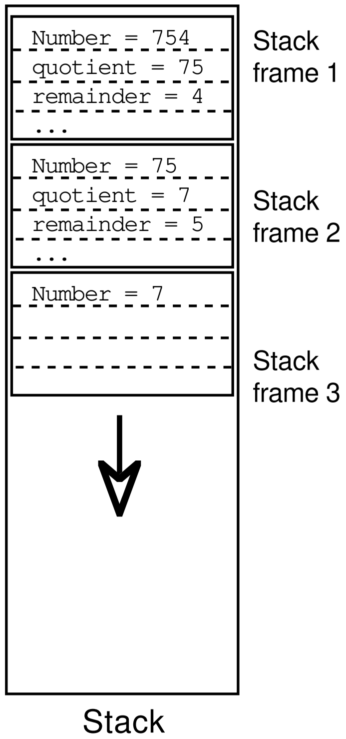

Figure 4.5 Stack during execution of recursive routine.

Nested structure

A FORTRAN program was always thought of as being made up from separately compiled files that got linked together. As noted earlier (section 4.4), this scheme has advantages like making it easy to develop subroutine libraries, but also disadvantages like the compiler being unable to check whether the calls to other routines were correct with respect to arguments etc.

An Algol program was a single composite entity. It contained inside itself a "main" routine and all the other functions and subroutines ("procedures") that it needed. With an Algol program you would have something like the following:

```
program demo;             Program header and
var                       declaration of shared
    integer count;        data variables
    …
    …
    procedure HeatFlow(…);
    begin                 Definitions of subroutines
           …              and functions
    end;
```

```
    procedure PrintResults;
    begin
            …
    end;

begin                       Main program
    …
    HeatFlow(…);
    …
    PrintResults;
end.
```

With only a single program file, the compilation process was simpler than that for FORTRAN. Figure 4.6 illustrates the approach.

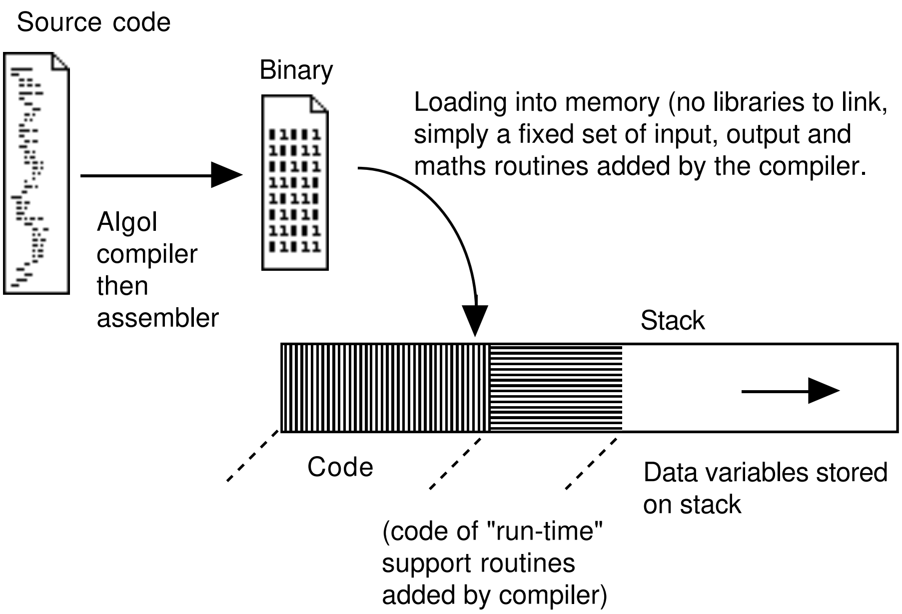

Figure 4.6 Algol style compilation and loading processes.

Compile time checks

This organization of a program made it possible for the compiler to do more checking of the code. A procedure declaration or function declaration had to specify details of the data values that that procedure would need to be given to work with (like the integer Number in the PrintPosNumber() function above). Later in the code, there would be calls to that procedure. The compiler could check that these calls provided appropriate data values.

Nesting could be taken further. A procedure declared within the main program could have other procedures and functions declared within it. These enclosed functions would be auxiliary routines used only within the procedure wherein they were defined. Usually there were limits on the extent of nesting, you couldn't define a procedure inside a function inside a procedure inside another function .….

This nesting scheme was meant to encourage and help programmers break down complex processing steps into lots of small functions that could be grouped together. However, it did make things more difficult for compiler writers to handle the source code and some aspects of the scheme forced the generation of less efficient machine code. Although some of the Algol family languages continue using nested declarations, other descendants of Algol (e.g. C and C++) have abandoned nested procedures.

Use of language

Never a great success as a practical programming language, Algol's importance is that it was the starting point from which most of the modern programming languages have evolved.

4.9 THE "ALGOL FAMILY"

4.9.1 AlgolW, Pascal, and Modula2

Niklaus Wirth, a Swiss computer scientist, is responsible for these members of the Algol family.

AlgolW

Along with a number of others at Stanford, Wirth developed AlgolW about 1965. Although it was meant to become a practical programming language for writing things like operating systems, compilers, etc., AlgolW was never taken further than a first limited version. This version was well suited to teaching programming, but was only available for IBM-360 computers.

Pascal compiler and its P-code interpreter

Wirth returned to Switzerland about 1970. In Switzerland, he only had access to a CDC computer, so he couldn't use AlgolW. He revised the language and wrote a compiler that generated partially interpreted code. The interpreter for this "P-code" was easy to implement in assembly language and versions were prepared for many different computers. This meant that the new language, Pascal, was soon widely available. But it was a fairly limited implementation and its main use was still teaching.

Modula2 was Wirth's third revision. It tidies up a few problem areas of Pascal and includes some minor extensions.

Although these languages have many small differences, they are basically the same, so the most commonly used, Pascal, can serve as an example.

Wirth used Algol as the base for these languages, so they have a nested structure, and they utilize stacks for organizing "automatic" memory space. Wirth omitted a number of features originally present in Algol that had proven difficult to use. He also improved the language with:

- more built in data types
- better control structures for iteration and selection
- programmer defined records

- dynamic storage structures

More built in data types

Boolean, character, string, and set data types

Algol had only had real and integer numbers. It is often useful to have other data types: "booleans" (variables that have the values `TRUE` or `FALSE`), characters (and groups of characters, or "strings", that can represent words or sentences), and "sets". All these were provided in AlgolW and its successors.

Control structures

Statements defining iterative (loop) structures

Of course, these languages have counting loops e.g..:

```
for i:= 1 to 10 do begin
..
    stuff to be done ten times
...
end;
```

But they have other loop constructs such as "*while*" loops ...

```
readln(value);            Gets first data value from user
while(value>0) do begin
    ...
                          Process positive data values
    ...
    readln(value);        Reads next value entered by user
    end;
```

Selection statements

For selection, there is an "if ... then ..." statement that can be used to choose whether an action is (or is not) performed:

```
if(current>maxfound) then
    maxfound := current; updates 'maxfound' if necessary
```

There is an "if ... then ... else ..." statement for selecting between two alternatives:

```
if(sex = 'F') then females := females+1;
else males := males+1;
```

Such statements can be concatenated to select from among more than two alternatives:

```
if(age < 1) then infants := infants+1;
else
if(age < 16) then children := children+1;
else adults := adults+1;
```

But there is an alternative statement for dealing with multiway selection that is often more convenient e.g.:

```
        case MonthNumber of
    1:          writeln("January has 31 days");
    2:          writeln("February usually has 28 days");
                ...
    12:         writeln("December has 31 days");
        end;
```

Records

Programmer defined "record structures"

AlgolW added record structures (inspired by the records used earlier in languages like COBOL); these records were further refined in Pascal and Modula2. So, if a programmer wanted to represent a "patient" in a program handling hospital records, it was possible to define such a record:

```
    type
        patient = record
                name : array [1..30] of char;
                age : integer;
                sex : char;
                ward : integer;
                ...
        end;
```

and then have variables of type patient (and also files of patient records).

Dynamic ("heap based") data structures

Dynamic data structures

The AlgolW language group also adopted ideas from Lisp. In these languages, it is possible to create structures "dynamically" when a program finds that they are necessary (just like in Lisp where lists and other structures are built as needed).

The "heap"

Another area of memory is reserved for these dynamic structures (it is called the *heap*). Space for dynamic structures can be requested in this area. Dynamic structures remain in existence until explicitly freed (when the space they occupied should be released).

Limitations

Pascal never quite made it as a full scale programming language (and Modula2 is of quite minor importance).

"Pascal is inefficient." This was often given as a reason for not using Pascal. The first compilers for Pascal were usually written to run quickly but to generate simple non-optimized code. (The standard compiler that generated interpreted code was used to move Pascal to a new machine. Once it was running on that the new machine, the code generation parts were changed to produce real assembly language rather than interpreted P-code.) Usually, Pascal was adopted first for teaching, and students typically spend much more time running the compiler than ever running their own programs so efficiency of programs wasn't that important. The nested procedural structure does entail some run-time overheads that reduce efficiency, but this is not that major a problem. It is possible to get Pascal compilers that generate efficient code; it is just that most compilers don't try.

A more serious problem with Pascal related to the issue of separate compilation. The standard Algol structure of a single program with no separately compiled parts is far too restrictive. In fact it had to be abandoned. Practical programming projects need to make use of subroutine libraries and need to have schemes for separate compilation of program parts followed by linking (as in FORTRAN). Most implementations of Pascal allowed for separate compilation, but this involved extensions to the language that were non-standard (the standard has finally been revised to accommodate separate compilation). Compilers from different suppliers tended to provide such extensions in slightly different ways, so limiting the interoperability of Pascal code.

If you have separately compiled parts, then you have the problem of the compiler not being able to check consistency of these parts. This problem was solved in the extended Pascal dialects (and Modula2) using an approach developed earlier for BCPL and C (see section 4.9.3).

Another common complaint about Pascal was that "*Pascal makes you say 'please'*". The Pascal language has rather strict rules on how you use data and limits on the extent to which you can do things like find the addresses of data elements. These rules and limits are very helpful to beginners who tend to misuse data and to do things with addresses that really shouldn't be done! But in more advanced work, programmers often need to build up complex network structures that involve address data. In Pascal, code becomes a little verbose because everywhere data are used in a slightly non-standard way the programmer has to indicate that this was intentional ("saying 'please' to the compiler"). Many programmers disliked this style.

4.9.2 ADA

ADA is a rather specialized descendant of Pascal.

The ADA language was designed for the US Department of Defence for "embedded systems" (control programs in everything: fighter aircraft, radar units, automated warehouses, …).

The style of the language is vaguely reminiscent of Pascal, and like Pascal the compiler incorporates lots of analysis and checking to try to detect errors in a source program. But ADA adds much to Pascal: support for separate compilation, features to allow for multiprocessors, communications, mechanisms for handling run-time errors. The extensions are so numerous as to make the ADA language one of the larger, more complex languages now in use.

Defence related projects are often required to use ADA, but the language is otherwise not that popular.

4.9.3 BCPL, C, AND C++

Way back around 1963, an attempt was made to define a Combined Programming Language (CPL) that would somehow combine the "best" features of an algorithmic language like Algol and a data processing language like COBOL. This exercise proved to be too ambitious and was abandoned. (Later in the 1960s, IBM sponsored another attempt along these lines, combining features from FORTRAN, Algol, and COBOL; this project was completed and resulted in the language PL1.)

BCPL – a limited language designed for writing software systems

Martin Richards, a junior in the CPL project, invented BCPL (Basic CPL) a much simpler language with a much more specific aim. BCPL wasn't intended to be good at everything. Its aim was simply to be a good language for writing system's software (things like compilers, editors, components of operating systems etc). BCPL was moderately successful in this role. It did have limitations (e.g. no real numbers, but why would you want real numbers if you are writing something like a "device driver" (code to control a peripheral device)?).

The BCPL compiler was written largely in BCPL (with just a bit of assembly language code that could be easily changed). It translated BCPL statements into instruction sequences using little templates that were defined in terms of the instructions available on a particular kind of computer. If you changed these templates a BCPL compiler on one machine could generate code for a different kind of computer. This made the BCPL language easy to transport and quite a number of organizations had people using BCPL in the late 1960s.

The language "B"

Kernighan and Ritchie at ATT laboratories had a BCPL compiler. They decided to rework the language, removing limitations, adding a few features, while keeping the main idea of a language that would be good for writing system's software. Their first attempt (the language "B") was simply a minor reworking, simplification, and shortening of BCPL. Their second version was "C".

Caution, C, handle with care

Just a little language to call their own. Something suited to a couple of gifted, experienced programmers working on their own high tech projects. But C escaped. The world's first computer virus. C got out from the ATT laboratories and infected machines all over the world before it was ever made safe for the average programmer.

C was designed to be a language that would be suitable for writing system's software, like the core parts of an operating system. The code generated by the compiler had to be very efficient if the language was to be used in this way. Anything a compiler would have difficulties with was dropped. Consequently, in some respects the language is simpler than other Algol family languages that have retained the relatively complex nested program structures.

Down to the hardware level!

If C was to be used for writing things like "device driver code" (the code that actually interacts with the peripheral controllers), then it had to allow the programmer get down the hardware level and manipulate bits in specific registers and in particular memory addresses. This gives the programmer considerable power, and lots of responsibility.

The programmer is always right

The most dangerous design aspect of C was the requirement that the compiler should always assume that the programmer was always right. Even if a compiler could detect a probable error in a program, it should just go ahead and generate the code. After all, the programmer was going to be one of the ATT team (Thompson, Ritchie, or Kernighan) and they'd know what they were doing. They wouldn't need a compiler to second guess them or try to hold their hand and protect them..

All this makes C a rather dangerous tool for the average programmer. (If the Pascal compiler keeps making you say "please", a C compiler keeps making you say "sorry".) The power and the danger have attracted programmers and C is now one of the three main languages (C for programmers, FORTRAN for engineers, COBOL for accountants).

Actually, the ANSI standard for C put a lot of compile time checks into the language, so taming it a little.

C program structure

C programs are typically built from multiple, separately compilable source files, and functions taken from libraries. The program generation process is illustrated in Figure 4.7.

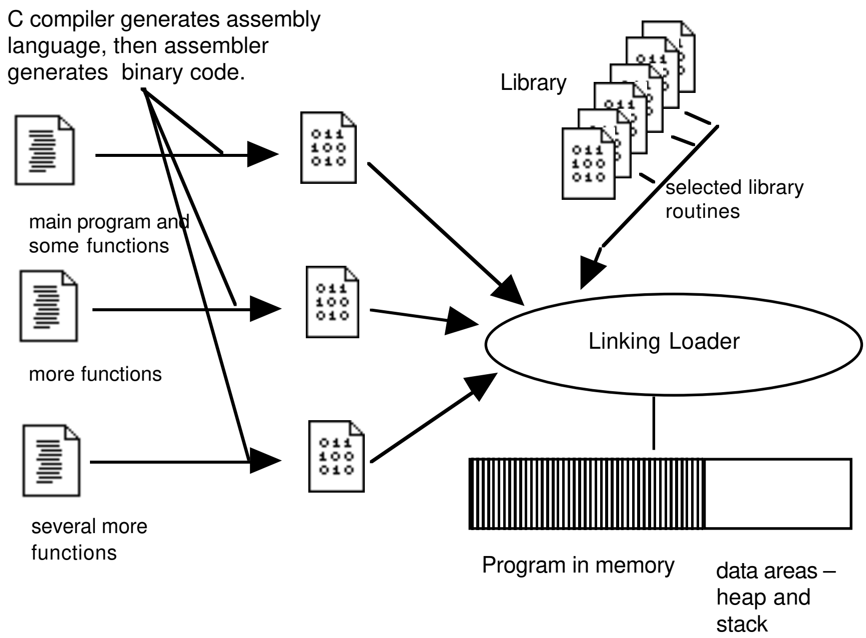

Figure 4.7 Compiling, assembling, and link-loading a C program.

This structure gives C the same advantages as FORTRAN

- it is easy to build up libraries of functions that can be used in many programs;
- a large program can be split into small parts with different programmers working on each part;

- the editing/compilation process tends to be efficient, you only need to recompile the file(s) containing code that has changed.

C provides a mechanism that a programmer can use to get the compiler to check the consistency of separately compiled parts of a program (you aren't forced to use this mechanism all the time, a C compiler will allow you to make as many mistakes as you want).

Header files

The source code for C routines is stored in files, usually these will have names that end in .c; e.g. File1.c, and File2.c. A second file, a "header" file, may be associated with each .c file; e.g. File2.c can have an associated header file File2.h. A header file contains information describing the routines (and any shareable data structures) that are defined in the associated .c file. Libraries, that contain very large numbers of functions, will have associated header files that describe their contents.

Using header information for compile time checks

Suppose the main program in File1.c uses functions from File2.c and you want the compiler to check that it is using them correctly. You can put a line into File1.c that tells the C compiler to read File2.h so that it knows what is defined in File2.c.

#include

This instruction to the compiler would say something like:

```
#include "File2.h"
```

and would appear on a line near the start of File1.c.

When it encounters a `#include` statement, a C compiler reads the information in the file specified and records details for later reference. If it later finds references to things defined in File2.c, it uses the information it saved to check that routines are being used correctly.

"Standard" header files

You will get used to seeing `#include` statements at the start of most files of C code. Usually, these will be 'including' header files that describe things in the standard libraries of input/ output routines or the mathematical functions.

A C source file will contain a number of routines ("functions" that compute values, and "procedures" that perform tasks). There may also be some definitions of data variables that can be shared by many routines (as in FORTRAN). One of the files forming a program has t contain a function called "main". This is the main program that contains the calls to all the other routines; the compiler arranges for this routine to be started when the program gets loaded into memory.

```
#include <stdlib.h>       Include standard header files
...


int SharedCounter;        Define any shared data
...

void Sort(int a[], int n)
{                         Define functions
   ...
}

int LoadData()
```

```
{
    ...
}

int main()
{                          Provide a main program
    ...
}
```

Each individual routine (function or procedure) is built up from sequences of statements: assignment statements for calculations, conditional statements for selecting among choices, iterative statements for loops, and "subroutine call" statements that invoke other routines.

All the modern (post-AlgolW) languages have essentially the same kinds of statement. There are differences in layout and in the keywords used, but the statements are essentially the same. Those in C are typically a little more concise than the equivalent statements in other languages like Pascal.

Statements

So, C has "for" loop and "while" loops; it has "if (...) ..." statements, and "if (...) ... else ..." statements; it has a multiway selection statement ("switch() { ... }"). Once you've learnt any one of these Algol family languages, you've really learnt them all!

Standard data types

The data types are again similar. There are the standard integer and real numbers, and characters. Unlike Pascal, C doesn't define any standard "boolean" or "set" data types (but it is easy to define your own if you need these data types).

Record structures

C has record structure definitions similar to those in Pascal (maybe slightly more flexible than those in Pascal). Of course a different syntax is used to define record structures.

C allows the programmer to chose how storage space is allocated to variables.

Static variables

One can have static data like FORTRAN. Space for static data is organized by the compiler and linker-loader and is allocated before the program starts and remains allocated the entire time that the program is running.

Automatic variables

As an Algol-family language, C uses stack-based "automatic" storage for most variables.

Dynamic, heap-based, variables

C also has dynamic storage with the program requesting space for structures that are built as the program runs (and freeing the space when the structures are no longer necessary).

In your first programming subject, you won't bother that much about the different kinds of storage, you will mainly use automatic storage. It is in your second programming subject that you will need to use all the different kinds of storage.

C++

You will actually be learning C++ rather than standard C.

C dates from around about 1970. C++ started as a dialect of C around about 1980, it has been revised twice since then. But they are still very similar languages (in fact, a correct C program should be acceptable to a C++ compiler).

The C++ language aimed to achieve three things:

- to be a better C
- to support "abstract data types"
- to permit the use of a programming technique known as "object oriented programming" (OOP).

A better C? C++ was designed to permit more compile time checking and also to offer alternatives to various features of C that were known to be common sources of programming errors.

"Abstract data types" – these will be the main topic of your second programming course, they are a kind of elaboration of the idea of a record structure.

OOP –see next section of Simula, Smalltalk, and Eiffel.

4.9.4 SIMULA, SMALLTALK, AND EIFFEL

These are the principal specialized "Object Oriented" languages.

Simula

Simula was developed in the mid-1960s as a language for simulations (simulations of anything you wanted: modelling aircraft movements at an airport, modelling the spread of a disease in a population of individuals, modelling the activities in an automated car wash, ...)

Objects

Simula was based on Algol-60 but added a variety of constructs that were needed for simulation work. Essentially, it allowed the programmer to create in the computer a set of "objects" (each of which owned some resources and had specified behaviours) that modelled things in the real world. Once the objects had been created, the Simula run-time system could mimic the passage of time and could allow the programmer to track interactions among the objects.

Smalltalk

The Smalltalk language was developed by the very innovative research group at Xerox's Palo Alto Research Centre (the same group as invented the prototype for the "Macintosh/Windows" OS and interface). Smalltalk offers a different way of thinking about programming problems.

Reusable component libraries

Usually, each problem is treated as if it were totally new. The problem gets analyzed, broken down into subtasks, and then new code is written to handle each of these subtasks. Smalltalk encourages an alternative view; instead of writing new special purpose code, try to find a way of building up a solution to a problem by combining reusable components.

The reusable components are Smalltalk objects. A Smalltalk system provides hundreds of different kinds (classes) of "off the shelf " reusable components. Actually, Smalltalk is an interpretive system (a bit like Lisp) and the language is not strictly in the Algol family.

Eiffel

In some respects, Eiffel is the best programming language currently available. It takes advantage of the experience gained with earlier languages like Simula, Pascal, Smalltalk, ADA and others.

It is a compiled language (so Eiffel programs are much more efficient than interpreted Smalltalk programs). The basic idea is the same as Smalltalk, i.e. the best way to construct programs is to build them out of reusable objects.

Although in many respects very good, Eiffel is restrictive. It enforces the use of an Object Oriented (OO) style. You have to learn several styles, not just OO. For this reason, you are learning C++ because it supports conventional procedural style as well as OO.

EXERCISES

1 Began programming on the electromechanical computing devices used in the 1940s by the US Navy to generate gunnery tables. Reputedly invented the term "bug" for a programming error after having had to remove a crushed moth that was jamming one of the relays in this computer. Rose to be a high ranking US Navy officer. Prime mover in the standardisation efforts that lead to the development of the COBOL programming language.

Identify this person and write a more complete biography.

2 Implemented one of the first (if not the very first) Algol compiler. Created the first operating system that was designed on a layered model with a kernel providing low level services, and surrounding layers that added functionality. Proponent of structured programming techniques and critic of earlier coding styles with their cris-crossing flows of control induced by indiscriminate use of "GOTO" statements. Tried to convince programmers that they needed a little discipline.

Identify this person and write a more complete biography.