$FA05, giving us a table of three two-byte pointers. Now when our imaginary interpreter encounters a token — $81, for example — it proceeds to subtract $80 from it, multiplies the result by two, and adds that to $FA00. The final result in this case is $FA02, which is the lo byte of the pointer to the PRINT subroutine. If a token other than $81 had been encountered, then the algorithm described would have returned the pointer address for the corresponding subroutine. In this way the BASIC command PRINT is replaced by a token, $81, which is an offset to a table of pointers that direct the interpreter to the relevant part of its own program.

That's a measure of the 'distance' between BASIC, a so-called high-level language, and machine-code, the low-level language. BASIC looks comprehensible to us because it uses English language code words, algebraic logic and numbers and strings. When we replace the words by tokens, and the rest by ASCII codes, it begins to look a lot more like something a microprocessor can handle — and, as we've seen with tokens, that's almost exactly the case.

The final thing to consider about memory manipulation is the idea of *context*. We've seen in the BASIC Text Area the widespread use of codes — ASCII codes to represent characters and numbers, tokens to represent commands, and (in the Spectrum) special binary codes to represent numeric data (see page 78). All of these codes reduce to binary numbers in the range 00000000 to 11111111 ($00 to $FF, 0 to 255 decimal) contained in single bytes of memory, and interpreted according to their context. In the BASIC Text Area of the Commodore 64, for example, the BASIC program line:

```
200 rem*******leftS*********
```

might have three bytes containing the decimal number 200 — once in the link address lo byte, once in the line number lo byte, and once in the token representation of 'leftS'. Each byte looks the same as the others, yet means something different. It is only your expectations that tell you how to interpret that value in different places.

This is really where we came in, at the start of the Machine Code course. Then we said that everything stored in a computer is in some sort of machine code. Some of this was familiar (like ASCII codes), some unfamiliar (such as tokens), and some as yet unexplained (such as machine code programs). So let's now start looking at machine code programs themselves.

OPERATION CODES

Programs in machine code are sequences of bytes located anywhere in memory that are a mixture of instructions to the microprocessor, and data for the microprocessor to operate upon. As with all other bytes of memory, it is only the context that can separate the data bytes from the instruction bytes, so we must first consider the format of machine code program instructions.

A machine code instruction begins with a code that identifies the operation to be performed. This is called the *op-code*, or *opc*, and may be one or two bytes in length. The op-code may be a self-sufficient instruction requiring no data, but more usually it is followed by one or two bytes of data. A single byte of data is likely to be a numerical constant or an ASCII code, while two bytes of data following an op-code are always an address (always stored in lo byte/hi byte form). With the above definition we immediately come upon differences between microprocessors: the BBC Micro uses a MOS Tech 6502A, the Commodore 64 uses a MOS Tech 6510 (very similar to the 6502A, so in future we'll talk generally about the 6502 only), and the Spectrum has the Zilog Z80A. MOS Tech and Zilog developed their microprocessors at about the same time — the early 1970s — following the release by Intel of the first microprocessor in 1971. Both the 6502 and Z80 therefore share a design philosophy, but they differ sharply in detail. In particular, Z80 machine codes are completely different from 6502 machine codes. Thus, for example, 6502 op-codes are always one byte long, and may be followed by one or two data bytes or by none; but Z80 op-codes can be two bytes long, followed by one or two data bytes or by none.

When sent to the microprocessor, an op-code is decoded by the CPU's internal program into operation and length codes, and it is this latter information that enables the microprocessor to interpret the bytes following the opc. For example, to the 6502 the sequence of hex bytes:

```
A9 0E 8D 01 4E 60 44 52 41 54
```

represents three instructions, followed by four bytes of ASCII codes. This could be re-written as:

```
A9 0E
8D 01 4E
60
44
52
41
54
```

showing that the first instruction is opc A9, which is always followed by one data byte; the next instruction is opc 8D, which is always followed by two data bytes; while the next is opc 60 which requires no data and causes program execution to branch, so that the following data bytes are not examined by the processor at all. If the microprocessor is sent the first byte, A9, when it is expecting to receive an opc, then everything will function smoothly thereafter. The information in each opc will ensure that the correct number of data bytes for each opc is picked up by the processor, and the following byte will be treated as the next opc. If, however, the processor is expecting an opc and is sent the second byte, 0E, then it will treat this as an opc, with the result that the sequence will be interpreted thus: