June 2006

Programming Then and Now: From the LGP-30 to the Laptop

Keith Smillie

Vulgar languages that want
Words, and sweetness, and be scant
Of true measure,
...
He that first invented thee,
May his joints tormented be,
Cramp'd for ever;
Ben Jonson (1573 – 1637),
A Fit of Rhyme Against Rhyme.

The LGP-30, the University of Alberta's first computer, was delivered in October 1957 and was soon used extensively for teaching and research throughout the University. It was replaced by an IBM 1620 in May 1961, but continued to be used for teaching for another two years. We shall use the LGP-30 as a starting point for a brief look at the evolution of computing in the Department of Computing Science at the University of Alberta. We describe the LGP-30 and its programming in some detail, giving some sample programs which have run on an LGP-30 simulator. We shall then give a very brief summary of the various computers that have been used in the Department subsequent to the LGP-30. Then we shall discuss the various programming languages that have ben used in introductory courses in the Department. To provide some continuity in this discussion and a basis for a comparison of the various languages we shall give an example, in a couple of instances two examples, of a program in each language for the problem of finding the frequency distribution of the faces which occur when an unbiased die with an arbitrary number of sides is thrown an arbitrary number of times.

However, to emphasize that computing did not begin at the University of Alberta with the LGP-30, we shall begin with a few notes on the means of computing, or more prosaically simply the means of doing arithmetic, in the period since the University was founded in 1908 until the introduction of the LGP-30 almost exactly fifty years later.

Slide rules, mathematical tables and calculators

Apart from hand calculations, the principal means of calculation in the period before the electronic computer were slide rules, mathematical tables, and desk calculators, or some combination of these aids. Slide rules, which gave only about three-figure accuracy, were used by many faculty and students throughout the campus. There was one 42-inch slide rule which hung for many years in the University Bookstore over the counter where the almost universal ten-inch slide rules were sold. It is now part of a permanent exhibit of early calculating devices and machines just outside the offices of the Faculty of Science in the Biological Sciences Building.

For many years the Department of Mathematics used a set of mathematical tables prepared by Professor J. W. Campbell who came to the University in 1920. These tables contained the usual tables for the common logarithm, square, cube and reciprocal, circular and exponential functions, as well as an extensive table of the hyperbolic function which Professor Campbell "himself calculated on a hand machine". These tables were first published locally in 1929, and were reprinted in 1946. At some subsequent date they were replaced by *Knott's Four-Figure Mathematical Tables*, first published in 1900, and were used until the mid 1960s. The front cover bore the following warning printed in red: "THIS BOOK IS THE PROPERTY OF THE UNIVERSITY OF ALBERTA. It must not be defaced or mutilated. Private possession may be cause for action by the University authorities." They were eventually remaindered in the University Bookstore for twenty-five cents each. The only other mathematical tables of which there appears to be any record are the eight-place Vega tables used in the Faculty of Engineering. Of couse, many textbooks contained in appendices a selection of tables related to the subject treated in the text.

A variety of desk calculators, both hand-operated and electrically driven, were used throughout the University. In the Department of Mathematics there were hand-operated Munroe and Multo calculators as well as a couple of electrically driven calculators. There was also the Swedish-built Odhner calculator weighing thirteen pounds in which multiplication and division were performed by addition and subtraction with repeated shifting. The cover of the small instruction booklet had a picture of a tastefully dressed young woman displaying the calculator, together with the following text: "Let Original-Odhner do the work for you. Learn to count the modern way on the new, attractively styled Original-Odhner." In the Faculty of Agriculture the Munromatic was used for statistical calculations since it allowed the simultaneous calculation of sums, and sums of squares and cross-products as well as, by means of a counter attached to the side of the machine, the number of sets of data being accumulated.

Some texts had sections on the use of desk calculators. One we shall mention is the two-volume statistics text co-authored by Professor E. S. Keeping of the Department of Mathematics. In the introductory chapter of the first volume there is a short section entitled "Calculating Machines" in which we read the following: "A calculating machine is constructed to add and subtract. By means of continued addition and subtraction, operations involving multiplication, division and square root can also be performed with great speed." Later in the same section he gives details for performing repetitive calculations using as an example finding the values of "$12 + 6x$ for $x = 5, 7, 15, 12$, etc." (In 1961 the

Department of Mathematics moved into the newly opened Physical Sciences Centre. The booklet commemorating the official opening had a photograph of the computing room showing a number of well-dressed middle-aged faculty members, including Professor Keeping, seated at tables using desk calculators.)

Finally we shall mention the Millionaire Calculator, a large and cumbersome machine measuring 25 inches by 12 inches by 6 inches and weighing up to 120 pounds, which was first produced in Switzerland in the mid 1890s. Between 1894 and 1935 a total of 4655 Millionaires were sold in Europe and the United States with government agencies being the largest customers. The title page of the brochure advertised "Calculating machines of superior workmanship, embracing Expedition and Accuracy in reading results in the Four Rules of Arithmetic, Economy in Time and Energy of the Operator." The following page said that "[It] is the most efficient Calculating Machine in the world. Requires only one turn of the crank for each figure in the Multiplier." For many years there was a Millionaire in a display case in the Department of Mathematics. How it arrived there or whether it was ever used is not known. It is now part of the calculator exhibit in the Faculty of Science. Beside it is a key ring with a calculator about one inch square which can perform addition, subtraction, multiplication, division, and square root.

The University of Toronto FERUT

The first use of an electronic computer on the campus was in the Department of Physics which in May 1957, a few months before the LGP-30 was purchased, established a teletype connection with the FERUT, Ferranti Computer at the University of Toronto. The teletype machine was located in what was termed a "glorified closet" in the basement of the Arts Building and a connection was established on Tuesday evenings. The system was said to have worked tolerably well except when there was a thunderstorm anywhere between Edmonton and Toronto.

FERUT was a fixed-point one-address binary computer with a most idiosyncratic order code. One example in the *Programmers' Handbook* had the following program fragment to replace a number stored in memory location `/ C` by its cube:

```
/
E  I S T /
@  / C / K
A  / C / F
:  / C / A
S  / C / F
I  / C / A
```

(One user reputedly remarked that the preponderance of the symbol `/` in machine-language programs was reminiscent of the rain in Manchester where the computer had been designed and built.)

It is no wonder that the staff of the Computing Centre at the University of Toronto developed Transcode which enabled FERUT to be progrmmaed as a floating-point, three-address decimal machine with a very much simplfied order code. Patterson Hume, one of the developers of Transcode, recently said that "It literally changed everything overnight, whereby creating a computer language so that people who work with code could commumicate in English (i.e. their natural language)."

The LGP-30

The LGP-30 was ordered in September 1957 and arrived the following month. The original installation consisted of the computer with a Flexowriter console (a modified electric typewriter with a mechanical paper tape reader and punch), a photoelectric paper tape reader and a mechanical paper tape punch, and an additional Flexowriter for the preparation of program and data tapes. It was installed first in the basement of the Arts Building but was soon moved to the Arts Building Annex.

The LGP-30, which was termed a "desk computer" not because it could be placed on a desk but because it was the size of a desk, was 26 inches deep, 33 inches high and 44 inches long, and weighed about 800 pounds. One user described it as "a great hunk of a machine – about the size of a deep freeze." The main memory consisted of a magnetic drum with a capacity of 4096 32-bit words arranged in 64 tracks of 64 sectors each with one word per sector. There were three circulating registers: the Accumulator Register for arithmetic operations, the Instruction Register for holding the current instruction, and the Control Register for the address of the next instruction to be executed.

The internal operation was binary so that all data had to be converted from decimal to binary on input and from binary to decimal on output. Data were assumed to be represented in fixed-point form and less than unity in absolute vale. Negative numbers were represented as two's complements. The format of a word representing a number was the sign bit (0 for a positive number and 1 for a negative number) followed by 30 bits for the magnitude. The 32nd bit was normally not used in the representation of either numbers or instructions but was a "spacer" bit separating words on the magnetic drum.

There were very few reference manuals for the LGP-30. There was a glossy illustrated *Operations Manual* which was printed, and the following manuals which were typescript: *Programming Manual*, *Programming Class Notes*, and a manual entitled *Subroutine Manual Coding Sheets* giving various utility programs. There was also a manual for an "algebraic compiler and translator system". The *Programming Manual* consisted of fifty-six coil-bound pages written in understandable but rather unpolished prose. The introductory paragraph contained the following interesting assertion: "Programming the Royal Precision LGP-30 is basically simple. Understanding certain problems requires certain knowledge, however programming for the LGP-30 does not." As we give a very brief introduction to machine-language programming in the next section, the veracity of this assertion may well be questioned.

LGP-30 programming - A simple example

We shall introduce machine-language programming by discussing a program to find the sum of two arbitrary numbers. For test data we shall use the integers 5 and 7.5 which have the sum 12.5. We shall assume that the two numbers to be summed are already stored in memory and that the sum is to be stored in memory.

There are sixteen instructions for the LGP-30, only four of which will be used in this example. Each instruction has a one-address format giving the operation to be performed and the address, i.e., location in memory, of the single operand. As we noted in the last section, arithmetic in the LGP-30 is performed in the one-word Accumulator Register. Therefore the program for finding the sum may be stated in words as follows:

> *Bring* the first number to the Accumulator; *Add* the second number, and retain the sum in the Accumulator replacing the value of the operand which was previously there; store the sum in a specified word in memory and *Hold* the sum in the Accumulator; *Stop*.

Now we must assign locations in memory for the two numbers to be added and for their sum, and for the program. We shall assume somewhat arbitrarily that the first number is stored in location `3750`, i.e., on track `37` and sector `50`, and the second number in location `3751`. The program, which consists of four instructions, will be stored in locations `0100`, `0101`, `0102` and `0103`, and the required sum in location `0104`. We may now give the program in the conventional manner of showing machine-language programs with the instructions given one-per-line with the locations in memory on the left and comments on the right:

```
0100  b3750 Bring first number to Accumulator
0101  a3750 Add second number
0102  h0104 Store sum
0103  z0000 Stop
0104        Sum
```

Now let us see how this program is stored in the memory of the computer. Each instruction occupies one 31-bit word, where the bits are numbered 0, 1, 2, ..., 30. The operation is represented by bits 12, 13, 14 and 15, and the address of the operand by bits 18 to 29, inclusive. The following shows the contents of the four locations with addresses `100`, `101`, `102` and `103`:

```
0000000000000001001001011100100
0000000000001110001001011100110
0000000000001100000000010001000
0000000000000000000000000000000
```

The bits representing the instructions as well as those representing the addresses are in bold. We see that the operation *Bring* is represented by `0001` in binary which is 1 in decimal, *Add* by `1110` or 15, *Hold* by

`1100` or 12, and *Stop* by `0000` or 0. The address `3750` of the first operand is given by `100101110010` where the first six bits `100101` give the track and the second six `110010` the sector. This corresponds to the external representation of `37` for the track number and `50` for the sector. The addresses of the operands in the remaining three instructions may be interpreted similarly, and we note the address of the Stop instruction is not used.

We have said that all data stored in memory is in binary and that each number is considered to be less than unity in absolute value. This restriction applies to data that are input initially and to all numbers that are calculated by the program. The procedure of ensuring that these conditions are met throughout a computation is known as “scaling” and may be considered to be one of the most troublesome problems in machine-language programming. Conversion of data from decimal to binary on input and from binary to decimal on output was handled by subroutines in the small program library supplied with the LGP-30 and need not concern us here. However we shall consider the scaling required in our simple example of finding the sum of two numbers.

If the two numbers, 5 and 7.5, to be summed are each divided by 2^3 or 8, the quotients are 0.625 and 0.9375, respectively, so if converted to binary may be represented in memory. However, this scaling is not satisfactory since their sum is 1.5625 which is greater than 1. Thus scaling by a division by 2^4 or 16 is required giving addends of 0.3125 and 0.46875. The sum of these numbers is 0.78125 which when multiplied by the scale factor of 2^4 gives the required decimal sum of 12.5. This sum would be stored in location 0104 in memory as

```
0110010000000000000000000000000
```

representing the binary number

0.11001000...

which has the decimal value $2^{-1} + 2^{-2} + 2^{-5}$ or $0.5 + 0.25 + 0.03125$ or 0.78125.

In the next section we shall discuss the complete order code of sixteen instructions for the LGP-30 and also an LGP-30 machine-language simulator which was used to run a number of machine-language programs including those discussed in this paper. In the following section we give three short programs as further examples of LGP-30 programming in machine language. The first of these programs has been taken from the course notes for what is possibly the first programming course given at the University of Alberta, and the second is a variation of this program illustrating the use of subprograms.

The third example is a program for finding the frequency of occurrence of each of the faces when a die with an arbitrary number of sides is thrown an arbitrary number of times. We shall assume that the number of faces may be any number between 1 and 20, inclusive, keeping in mind that unbiased dice exist only for the five Platonic solids, i.e., the tetrahedron, cube, octahedron, dodecahedron and icosahedron with 4, 6, 8, 12 and 20 sides, respectively. This example will be used throughout the remainder of this paper in the discussion of the development of programming languages. For sample data we shall use the results

```
1 4 2 3 1 1 3 3 4 2 3 4
```

of throwing a four-sided die 12 times which gives the following frequency table with the face numbers in the first column and the frequncies in the second:

```
1  3
2  2
3  4
4  3
```

In many of the programs the list of data is preceded with an integer giving the number of faces on the die and ends with a zero acting as a flag indicating the end of the data. Therefore the sample data are often given as

```
4 1 4 2 3 1 1 3 3 4 2 3 4 0  .
```

The reader who is not interested in a further understanding of machine-language programming for the LGP-30 may safely skip the following two sections.

LGP-30 programming – Order code

We shall list the sixteen machine-language instructions for the LGP-30 giving for each its name, its one-letter, binary and decimal representations, and a brief description of its function. An instruction may be represented as `op addr`, where `op` represents the operation and `addr` the operand. Note that the definitions of the Input and Print instructions given here correspond to the implementation of these instructions in the LGP-30 simulator discussed at the end of this section rather than to their implementation in the LGP-30 computer.

Stop: `z 0000 0`

`z 0000` Stop computation.

Bring from Memory: `b 0001 1`

`b addr` Replace the contents of the Accumulator by the contents of location `addr`.

Store Address: `y 0010 2`

`y addr` Replace the contents of the address portion of memory location `addr` with the contents of the address portion of the Accumulator.

Return Address: `r 0011 3`

`r addr` Replace the address portion of memory location `addr` by one plus the value of the Counter Register.

Input: `i 0100 4` (See note at beginning of definitions)

`i 0000` Input one 7-bit character from the (simulated) tape to the Accumulator displacing the contents of the Accumulator 7 bits to the left with a maximum of 28 bits in the Accumulator..

Divide: `d 0101 5`

`d addr` Divide the number in the Accumulator by the number in memory location `addr` and store the quotient in the Accumulator.

N multiply: `n 0110 6`

`n addr` Multiply the number in the Accumulator by the number in memory location `addr` and store the least significant 31 digits of the product in the Accumulator.

M multiply: `m 0111 7`

`m addr` Multiply the number in the Accumulator by the number in memory location `addr` and store the appropriate sign bit and the most significant 30 digits of the product in the Accumulator.

Print: `p 1000 8` (See note at beginning of definitions)

`p addr` Append the contents of the Accumulator to the (simulated) printer.

Extract: `e 1001 9`

`e addr` Form the logical product of the contents of the Accumulator and the contents of memory location `addr` and store the result in the Accumulator.

Unconditional transfer: `u 1010 10`

`u addr` Replace the number in the Counter Register with `addr`.

Test: `t 1011 11`

`t addr` If the number in the Accumulator is negative, replace the number in the Counter Register by `addr`.

Hold and Store: `h 1100 12`

`h addr` Store the contents of the Accumulator in memory location `addr` and leave the contents of the Accumulator unchanged.

Clear and Store: `c 1101 13`

`c addr` Store the contents of the Accumulator in memory location `addr` and clear the contents of the Accumulator.

Add: `a 1110 14`

`a addr` Add the number in the Accumulator to the number in memory location `addr` and store the sum in the Accumulator.

Subtract: `s  1111  15`

`s addr` Subtract the number in the Accumulator from the number in memory location `addr` and store the difference in the Accumulator.

In order to run the machine-language programs given in the previous section and in the following section an LGP-30 simulator was written in the **J** language. This simulator is available at

www.cs.ualberta.ca/~smillie/Jpage/Jpage.html

which contains information on the **J** language as well as links to further information on **J**. Here we shall show only how to input, run and display the result of the sample program for finding the sum of two numbers given in the previous section. The following gives an annotated record of the processing of this problem:

```
NB. *** Sum of two numbers ***
NB. Program as a 4-item character list
   Prog1=: 'b3750';'a3751';'h0104';'z0000'
NB. Data with scale factors as a 2-item list
   Data1=: 5 4;7.5 4
NB. Reset memory
   Reset ''
NB. Load program into locations 100, 101, 102, 103
   100 Load Prog1
NB. Load data into locations 3750, 3751 and convert to binary
   3750 Load Data1
NB. Run program starting with instruction in location 100
   Run 100
NB. Convert to decimal and display unscaled sum stored in location 104
   4 Show 104
12.5
```

LGP-30 programming – Three examples

One of the first, if not the very first, programming course at the University of Alberta was given by Bill Adams in January and February of 1960. Bill was a part-time programmer in the Computing Centre and became one of the founding members of the Department of Computing Science when it was formed in April 1964. His course notes, hand-written on yellow paper, still exist and make interesting reading over forty-five years later.

The first programming example given in the notes was to find the area of the annular ring formed by two concentric circles of given radii. The required area is the area of the larger circle minus the area of the smaller circle, or $\pi R^2 - \pi r^2$ where R and r are the two radii. This expression may be written as

$\pi(R + r)\,(R - r)$

which is the form Bill used in his program shown below. We note that the scaled values of R, r and π are assumed to be stored in memory locations 0010, 0011 and 0009, respectively, and the sum in location 0012.

```
0000  b0010    Bring R to accumulator
0001  a0011    Add r
0002  c0012    Store R + r
0003  b0010    Bring R to accumulator
0004  s0011    Subtract r
0005  m0012    (R + r) × (R – r)
0006  m0009    pi × (R + r) × (R – r)
0007  c0012    Store Area of ring
0008  z0000    Stop
0009           3.14159          ) Scaled
0010           R                ) binary
0011           r                )
0012           Area of ring     )
```

Area of an annular ring

The second example is a variation of this problem where the required area is found as the difference of the areas of the two circles where the area is calculated in a subprogram. The main program occupies locations 0000 to 0013 and the subroutine locations 0100 to 0105. We note the use of the Return Address instruction in lines 0001 and 0005 of the main program to set the appropriate return links in line 0103 of the subprogram. The program and subprogram are shown on the next page.

The third example, as has been mentioned in a previous section, is finding the frequency of occurrence of each of the faces given the results of throwing an unbiased die with an arbitrary number of faces an arbitrary number of times. The program is shown as the first program in the Appendix which gives the programs for this example for each of the programming languages discussed in the following sections of this paper.

```
0000  b0011    Bring r to accumulator
0001  r0103    Set return link
0002  u0100    Transfer to subroutine
0003  c0012    Store area of smaller circle
0004  b0010    Bring R to accumultor
0005  r0103    Set return link
0006  u0100    Transfer to subroutine
0007  s0012    Subtract area of smaller circle
0008  h0013    Store area
0009  z0000
0010           R                      ) Scaled
0011           r                      ) binary
0012           Area of smaller circle )
0013           Area of ring           )
```

```
0100  h0105    Hold radius
0101  m0105    Square radius
0102  m0104    Multiply by 3.14159
0103  u0000    Return link
0104           3.14159                 Scaled binary
0105                                   Temporary storage
```

Area of an annular ring with subprogram

This is the last time we shall have to consider the troublesome problem of scaling to ensure that all data and all numbers arising during the computation are less than unity in absolute value. We note that the input data, i.e., the number of faces on the die and the results of throwing the die, are scaled with a scale factor of 5 so that, for example, a face value of 3 would be stored as

```
0000110000000000000000000000000
```

which would represent the value 3×2^{-5} or 0.09375. The frequencies, 3, 2, 4 and 3 in this example, are scaled for programming convenience with a scale factor of 29 and would appear in memory as follows:

```
0000000000000000000000000000110
0000000000000000000000000000100
0000000000000000000000000001000
0000000000000000000000000000110
```

LGP-30 programming – The ACT I compiler

To simplify programming for the LGP-30 the ACT I Compiler (Algebraic Compiler and Translator) was introduced in 1959. Unfortunately it arrived too late for any serious use at the University of Alberta, but Bill Adams recalled recently his "shock and disbelief that such a thing would work". The compiler was documented in a short eighteen-page manual which was simply but attractively produced. The body text was in Small Capitals while example programs and all other statements, phrases, names and symbols in the source language were in Regular font. The first paragraph of the Introduction is an excellent statement of the motivation for its development and the philosophy of its design:

> The Royal McBee ACT I Programming System was designed to make the programmer's job of coding as painless as possible. Programming information is presented to the LGP-30 in much the same manner as might be used to explain a problem to a colleague. The common basis for the exchange of mathematical ideas is the language of algebra. This common language of algebra is the basic language of the ACT I Programming System. It has been augmented somewhat beyond basic algebra to give the programmer the power to insert logical decisions into his program and to work with arrays or sets of parameters as readily as handling one variable or constant. This new language is known as the “source” language; it is this language that this manual deals with.

Although primitive when compared with programming languages today, it is a most remarkable accomplishment of its own day. It is easy to see the enthusiasm with which it was greeted when it was released especially by those who had experienced the drudgery of months or even years of machine-langauge programming.

We shall make only a few remarks on the language. The objects in the language with the modern designations in parentheses were *symbols* (variable names), *operations* (operators, functions, verbs), constants, *brackets* (parentheses), *statement symbols* (labels) and *region symbols* (array names). Symbols could have up to five alphanumeric characters and were limited to 127 in number; there were 63 primitive operations including the machine-language operations; constants were limited to 39 in number; and statement symbols were of the form s0, s1, ..., s255. The ACT I program for the dice-throwing example is given in the Appendix.

After the LGP-30

In this section we shall give a very brief discussion of the computer hardware that became available to the Department of Computing Science, in either the Computing Centre or the Department, since the acquisition of the LGP-30 in 1957. The main theme of this paper, which is the use of programming languages in the Department, is resumed in the following section.

As we noted at the beginning of this paper, the LGP-30 was replaced by an IBM 1620 in May 1961 although the LGP-30 was kept for teaching purposes for another two years. The original IBM 1620 was replaced later in the same year it was acquired by a larger 1620 system with a card reader and punch, and the original 1620 was acquired by the Alberta Research Council which had been using about forty percent of the available time on it. The 1620 was used on an open-shop basis, as had the LGP-30, with users debugging and running their own programs. However as demand increased a gradual change to a closed-shop operation was initiated in which card decks were submitted by users and then run by Computing Centre staff. On April 1, 1963 a completely closed-shop operation was instituted.

In order to meet the increasing demand for computing time an IBM 7040/1401 was installed during the summer of 1964 and the IBM 1620 was phased out during a three-month period. The new system consisted of a central processor with 32 768 words of core memory, six tape drives soon augmented by more tape drives, and peripheral input and output. Even this system proved inadequate to meet the demand and it was replaced in the summer of 1967 by an IBM 360/67 which itself was upgraded several times over the next few years. By November 1969 the system had a total of 768 000 bytes of core storage, a magnetic drum, two disk drives each with eight discs, eight tape drives, two card readers and punches, and fifty-six terminal ports allowing interactive time-sharing capability. The IBM 360/67 continued to be upgraded with additional printers and tape drives and by the end of 1974 was operating at full capacity.

The following year it was replaced with an Amdahl 470 V6 which was soon upgraded and then replaced with an Amdahl 580.

The IBM 360/67 and later systems could be accessed either by batch processing or interactive remote terminals. A student-oriented batch facility or SOBF was developed to allow student programs to be processed quickly and efficiently. Students were issued “SOB tickets”, each valid for one run. Those students using up their allowance of tickets could purchase “SOB balls”, each costing five cents and equivalent to one ticket, made of transparent plastic and dispensed by a gum-ball machine that had been purchased at an auction. During the 1979/80 academic year the SOB facility was replaced with batch entry from terminals, and punched-card entry of programs and data was no longer available much to the dismay of a few users.

During this rapid expansion of computing facilities available to the University there was a similar growth of facilities in the Department of Computing Science beginning in 1970. The first computer was a used Digital Equipment Corporation PDP-9 costing 19 700 dollars with 8K memory, teletype console and oscilloscope display, a paper tape reader and punch, and a card reader. This was the first of many PDP systems that were acquired by the Department. At first the Michigan Terminal System was used as an operating system but in 1973 this was replaced by the UNIX operating system, possibly the first use of UNIX outside of the Bell Laboratories where it had been developed. Other computers acquired by the Department included the Nanodata QM-1, VAX 11/780s which replaced the PDP computers in the early 1980s, and Sun-1 workstations.

The first desktop computer in the Department was the IBM 5100 purchased in 1975 for 18 300 dollars. It measured about 18 inches by 24 inches by 8 inches, weighed forty-eight pounds and fitted conveniently on the top shelf of an audio-visual trolley so that it could be easily moved between classroom and faculty offices. It was used for teaching and research with APL and to a limited extent for teaching BASIC. (The IBM 5100 was followed in 1981 by the IBM 5150 and two or three years later by the IBM 5155; these machines were better known as the IBM PC and IBM XT, respectively.) The Department of Computing Services installed its first microcomputer laboratory consisting of twelve IBM PCs in October 1982 and its second with twenty-eight PCs in 1984. The Apple Macintosh was announced in 1984 and in the same year four Macintosh laboratories with a total of 100 Macintoshes were opened. It was the Macintosh rather than the IBM Personal Computer that caught the fancy of most, but not all, of the academic staff of the Department of Computing Science which used these machines very enthusiastically for much of their teaching.

To not prolong this discussion of hardware we shall jump to the present time. Faculty have one or more computers in their offices, and probably a similar number at home. Most students have a personal computer and often carry them to campus. The laboratories in the Department of Computing Science have a total of approximately 230 work-station positions, and the small seminar rooms have facilities for visual presentations.

Finally how may we measure the evolution of hardware over the 50 years since the introduction of the LGP-30 in 1957? The LGP-30 was an 800-pound desk-sized computer costing 40 000 dollars in 1957 which is approximately 280 000 dollars today. It had a memory of 4096 32-bit words, i.e., 0.000016 GB. Multiplication time in one list of specifications is given as 24 000 microseconds. Software for the LGP-30 consisted of a few subprograms for input-ouput operations, floating-point arithmetic and some of the more common mathematical functions. Now one thousand dollars will purchase a laptop weighing only a few pounds with a 60 GB hard drive and a speed of 1.66 GHz. (Multiplication on a roughly comparable desktop was estimated at 0.02 microseconds.) Software for the laptop is an operating system and optional packages for word processing, photo editing, and whatever programming languages, if any, the user requires.

Programming languages

Now we turn to a very brief look at the programming languages that have been used in introductory courses in the Department of Computing Science. Of course some of these languages, especially Fortran, have also been used throughout the University for research purposes. We shall divide these langauges into two groups, In the first group are the conventional "word-at-a-time" languages where the basic item of computation is the individual number or character so that operations on lists, e.g., summing a list of numbers, is performed by handling the numbers in turn as they would be if they were summed using a pocket calculator. In this group we include Fortran, BASIC, Algol W, Pascal, C and C++, Java and Perl. The second group will consist of array languages in which the basic item of computation is the array, e.g., one-dimensional lists, two-dimensional tables, and in some languages arrays of arbitrary dimension. This group will include spreadsheets (which many persons would not consider to be languages), MATLAB, and APL and its modern dialect **J**.

For each of the languages mentioned above we shall make a few historical remarks, and then give, mostly in the Appendix, the dice frequency program for that language. For some languages we shall give more than one version of the program in order to illustrate how the language has evolved. We note that all of the progrms have been debugged, sometimes with help duly acknowledged at the end of the paper, except those for the LGP-30 ACT I compiler already given, the University of Waterloo Watfiv compiler for Fortran, and Algol W. Then we shall attempt to make some general comparisons of these programs by noting their similarities and differences, and give some suggestions for alternative ways of introducing programming languages in a computing science curriculum.

We shall conclude this section with a few remarks about two languages which were never used in introductory courses in the Department but which for different reasons may be appropriate to mention here. These two languages are PL/I and Algol 68.

PL/I is a language developed within IBM in the 1960s to serve as a general-purpose language for scientific, commercial and special-purpose applications on their System/360 computers. We cannot recall ever hearing the suggestion that we should consider its use in the Department. However, in the mid 1970s a very strongly worded letter filtered down from the highest echelons in the University administration supporting a complaint from the Edmonton offices of a large company that we were remiss in not teaching our students PL/I because it was being used in their company. One very firm, but polite, letter from the Department saying that it was not our purpose to teach what was immediately useful in the marketplace terminated our involvement in the discussion.

Algol 68, as the name implies, is in the Algol family of languages, and was designed and implemented over a number of years starting in the 1960s. It is a language which was designed by a committee whose membership was, initially at least, about two-thirds European, and which some people termed an unruly committee which finally broke up in disarray, We mention Algol 68 here only to pay tribute to a colleague, Barry Mailloux, who was one of the leading members of the committee and one of the authors through its many drafts of the *Final draft report on the algorithmic language ALGOL 68* published in 1968. Barry received an M.Sc. from the University of Alberta in 1963, and then went to the Mathematisch Centrum, Amsterdam where he wrote his doctoral dissertation on Algol 68. On the completion of this work in 1968 he joined the Department of Computing Science where he became noted for his enthusiasm for Algol 68 and for his unfailing amiability. Sadly, Barry died of a brain tumour in 1982.

Conventional languages

In 1954 IBM formed a small group headed by John Backus, to develop an automatic programming system for the IBM 704 that would produce efficient object code that would execute at a speed about the same as hand-generated code. The language was called Fortran for “Formula Translating (Language)”. Emphasis was on efficiency of compilation and execution rather than on the design of the language. Indeed Backus remarked in retrospect that they simply made up the language as they went along and did not regard language design as a difficult problem. Fortran I was available in 1956 for the IBM 704, and the first paper on the language appears to have been given at the Western Joint Computer Conference in 1957. In 1958 Fortran II was officially released for the 704 and shortly after for other IBM systems. Later versions of Fortran include Fortran IV released in 1964 and Fortran 77 and Fortran 90. Fortran was extensively used for teaching in the Department of Computing Science, and for research throughout the

University. The Appendix gives Fortran programs for the dice frequency problem written in the style of Fortran II and also of the University of Waterloo Watfiv that was used in the Department. In the very early 1970s Fortran was replaced in courses for computing science students by Algol W although it continued to be used for some years in service courses.

BASIC, for Beginner's All-purpose Symbolic Instruction Code, was developed at Dartmouth College, a small liberal arts college in Hanover, New Hampshire. (Incidentally, it was at a meeting of the American Mathematical Society at Dartmouth College in 1940 that George Stibitz of the Bell Telephone Laboratories first demonstrated the remote use of a computer over a communications line.) The BASIC language was developed as an alternative to Fortran for non-science students that would be "friendly", easy to learn and use, and convenient to access. It was designed by a group led by John G. Kemeny, who later became President of Dartmouth, and Thomas E. Kurtz. The first BASIC program under time-sharing was run at about 4:00 a.m. on May 1, 1964, and the first published program was the following:

```
10 LET   X = (7+8)/3
20 PRINT X
30 END
```

BASIC proved to be an extremely popular language and now exists in many versions. Visual BASIC released in 1991 provided a simple means of constructing graphical universal interfaces for using the language. BASIC may now be considered an almost universal language and has been made available on almost all computing systems. It has not been the most popular language in the Department but was used in computer literacy courses in the 1980s and 1990s. The two BASIC programs shown in the Appendix are in the style of the first BASIC and a more modern version and were both written using the QBASIC implementation of the language.

Algol, for "Algorithmic Language", was developed in Europe in the late 1950s as a universal programming language. It was described first in the Algol 58 draft report in 1958 and two years later in the final Algol 60 report. Algol W was similar in many ways to Algol 60 but contained many improvements. It was implemented at Stanford University on the IBM System 360. It was used in the introductory course for computing science students for several years in the 1970s. As was noted in the last section the Algol W program for the dice-throwing example has not been checked out since an Algol W compiler is no longer avaialable.

Pascal was originally developed by Niklaus Wirth of the Swiss Federal Institute of Technology (ETH) in Zurich as a language that would be efficient to implement and execute and that could be used for teaching the important concepts of computer programming. It was named after Blaise Pascal, the seventeenth-century French philosopher and mathematician. The language was also defined formally using Backus-Naur Notation which, some of us at least, thought should be introduced to students. There were various implementations of Pascal and several of these were used in various introductory computing courses.

The C programming language was developed at the Bell Telephone Laboratories in the early 1970s. It was derived from a language named B, which was derived from an earlier language BCPL. Originally C was designed as a systems language for the UNIX environment but was soon used for a large variety of applications. The C++ language, also developed at the Bell Telephone Laboratories, might be described as a superset of C with added features supporting classes. C and C++ have been used in introductory computing courses in the Department.

Java was originally designed for writing programs for computer chips embedded in electronic appliances. However, it soon was found ideal for the design and implementation of programs intended for distribution and use on the Internet. It is now the language introduced to first-year computing science students.

Perl, an acronym for "Practical Extraction and Report Language", was developed in the mid 1980s as a text-processing language for the easy and efficient manipulation of various types of files. It will be used for rhe first time as a first language in an introductory course in the Department in the Fall Term 2006.

Array languages

Although electronic spreadsheets became popular with the development of the VisiCalc program for microcomputers in the late 1970s, paper spreadsheets have been used by accountants for a very long time. The following two entries from *The Random House Dictionary of the English Language. Second Edition* (1987) may give some perspective:

> **spreadsheet**. 1. *Accounting*, a work sheet that is arranged in the manner of a mathematical matrix and contains a multicolumn analysis of related entries for easy reference on a single sheet. 2. *Computers*. See **electronic spreadsheet**.
>
> **electronic spreadsheet**, a type of software for microcomputers that offers the user a visual display of a simulated worksheet and the means of using it for financial plans, budgets, etc.

We should add to this second definition that any cell in an electronic spreadsheet may contain either a numerical value, or some text, or a formula so that its value depends on the contents of one or more other cells. As an example a cell might give the sum of the numbers given in a specified range of cells, and any change in these values would cause the sum to be changed appropriately. Spreadsheets have been scarcely mentioned in introductory computing science courses although some of the assignments intended for solution with one of the conventional languages would have been almost trivial if a spreadsheet had been used.

MATLAB, for “Matrix Laboratory”, was developed at the University of New Mexico and Stanford University in the late 1970s so that students could use some of the Fortran matrix subroutine packages without having to learn Fortran. MATLAB developed into an internationally used language for mathematical and statistical computations, modelling and simulation, visualization, and symbolic manipulation, to mention only a few of its applications. It has found only limited use in introductory courses given by the Department.

The APL language was developed by a native Albertan, Kenneth Iverson, as an alternative to conventional mathematical notation which could be implemented interactively on a computer. Unlike the conventional languages of the previous section, the data objects of APL are one-dimensional lists, two-dimensional tables, and rectangular arrays of arbitrary dimension. There is a large number of arithmetical and logical operations defined for these arrays as well as for individual numbers. On his retirement in 1991, Ken began the development of **J**, a “modern dialect” of APL which used the standard ASCII character set rather than the specialized symbols of APL and benefited from the many years of experience with APL in the very large community of users. APL was used enthusiastically by Bill Adams and myself for teaching and research, and with less enthusiasm and much less extensively by a few other members of the Department.

The Appendix gives programs for the dice frequencies example using the Microsoft Works spreadsheet, MATLAB, and **J** rather than in APL because of the convenience of the ASCII characters.

Another example

As another, and penultimate, example of the development of programming languages since the introduction of the LGP-30 we give at the end of the Appendix programs in LGP-30 machine language, BASIC and **J** for the simple problem of finding the minimum, maximum and sum of a list of an arbitrary number of positive numbers which could represent, say, book prices. In the LGP-30 program the data are assumed to be stored in locations `0050`, `0051`, ..., and the results in locations `0031`, `0032` and `0033`. In the BASIC program the data are given in the program and the the results are listed with a PRINT instruction. In each of these programs the end of the list of data is flagged by a zero price. In the **J** program the prices are given as the list `Prices` and there is no need to indicate the end of the list with a zero price.

We shall make a few remarks for the interested reader about the **J** program

```
Summary=: (<./,>./,+/)
```

as it undoubtedly appears somewhat cryptic to the uninitiated. In **J** the adverb / , which would be called an operator in most languages, may be considered as a generalization of the familiar summation operator Σ in conventional mathematical notation. Therefore if the list of book prices is given by

```
      Prices=: 20.00 10.99 23.95 25.95  24.00 23.95  ,
```

then the sum is `+/Prices`, and the minimum and maximum are `<./Prices` and `>./Prices`, respectively.

We might mention that the "insert" adverb `/` which has been used here monadically with a single argument has a dyadic form called "table" with two arguments. As an example of its use let the list

```
   D2=: 2 2 1 1 2 2 1 1 1 1
```

represent the results of tossing a coin 10 times, where the `1`s represent heads and the `2`s tails. Then the expression `1 2 =/ D2` gives the table

```
   0 0 1 1 0 0 1 1 1 1
   1 1 0 0 1 1 0 0 0 0
```

where the `1`s in the first row indicate the occurrence of heads on the 10 throws and the `1`s in the second row the location of tails. The expression `+/"1 (1 2 =/D2)` are the row sums `6 4` giving the frequencies of heads and tails. This brief discussion may help explain the terseness of the **J** dice frequencies program.

Comparison of languages

There are considerable differences between the languages in the three groups. Although we considered machine-language programming for only the LGP-30, the detail required in even simple programs – and their relative opaqueness when written - may be considered representative of most, if not all, machine-language programming. In contrast, the supression of detail possible with conventional languages and the use of natural language words in the type declarations and control structures and for variable names simplifies the structure of programs and significantly improves their readability when written. Array languages allow a further brevity of expression although the readability of the resulting programs is a matter which has been hotly disputed.

To illustrate again, and very briefly, the differences in programs written in the three categories of languages we shall consider the simple example of finding the maximum of a list of positive numbers which could represent, say, book prices. In the machine-language program and in the first BASIC program the list will terminate with a zero price indicating the end of the list, and in the second BASIC program the list will begin with an integer giving the number of prices in the list. For the array-language programs the list of prices will be given by itself.

The following is the LGP-30 machine-language program given as a 19-item list:

```
   +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
   |b0017|c0003|c0019|b0000|c0020|s0020|t0008|z0000|a0019|t0011|u0013|b0020|
   +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
      +-----+-----+-----+-----+-----+-----+-----+
      |h0019|b0003|a0018|h0003|u0003|b0100|z0001|
      +-----+-----+-----+-----+-----+-----+-----+
```

The following are two very similar BASIC programs which may be considered representative of the programs in conventional languages:

```
REM Maximum 1                  REM Maximum 2
DATA 20.00,10.99,23.95         DATA 6,20.00,10.99,23.95
DATA 25.95,24.00,23.95,0       DATA 25.95,24.00,23.95
Maximum = 0                    Maximum = 0
READ Price                     READ N
WHILE Price > 0                FOR I = 1 TO N
   IF Price > Maximum THEN        READ Price
      Maximum = Price             IF Price > Maximum THEN
   READ Price                        Maximum = Price
WEND                           NEXT I
PRINT Maximum                  Print Maximum
STOP                           STOP
END                            END
```

Assuming that the prices data have been entered into the spreadsheet or that the required variables have been defined, the programs for the (Microsoft Works) spreadsheet, MATLAB, APL and **J** are given by `=max(A1:A6)`if there are six prices, `max(Prices)`, ⌈*/PRICES* and `>./Prices`, respectively.

As great as are the differences in the programming languages between these three groups, we should mention some of the similarities and differences within the group of conventional languages. For example, repetition may be expressed in Fortran as `DO I = 1 , N`, in BASIC as `FOR I = 1 TO N`, in Algol W as `FOR I := 1 UNTIL N DO`, and in Java as `for (i = 1; i <= N; ++i)`. Repetition may also be expressed, for example, in Watfiv as `WHILE (D .GT. 0) DO`, in Algol W as `WHILE D > 0`, and in C as `while (d > 0)`. Algol W, Pascal, C and Java each has a block structure while Fortran and BASIC do not. Variable types cannot be declared in BASIC or in Perl, may be declared in Fortran, and must be declared in Algol W, Pascal, C and Java. Subprograms are called "functions" and "subroutines" in Fortran and "procedures" in Pascal. I maintain that these are differences of terminology, notation and syntax and not of concepts. Also experience in teaching these languages indicates that students have the same general problems with learning programming independent of the language. Only the details of the problems differ between languages.

In this discussion of the differences and similarities in the three groups of languages we have omitted any mention of the pleasure, even the wonderment, in being first introduced to a conventional language after having programmed in a machine language or to an array language after experience with conventional languages. We did quote, though, Bill Adams' expression of "shock and disbelief that [ACT I] would work". I had a similar experience in being introduced to Fortran after several years of programming in machine language (for the National Cash Register 102A and 102D). Bill and I experienced these emotions again in the mid 1960s when we were introduced to APL. Also I have always found pleasure in using a spreadsheet in the solution of a problem however simple. While writing this paper I have found it nostalgic, and occasionally frustrating, to relive some of these experiences as I went from machine language to conventional word-at-a-time languages and finally to array languages.

A first language

There are still at least two questions that must be resolved when faced with the variety of programming languages that are available. The first concerns the language we should use when introducing beginning students to programming. The second, and I believe even more important, question is *how* do we introduce students to programming. We shall address the first of these questions in this section, and the second in the following and concluding section of the paper.

In the past there have been those who have argued that we begin either with a machine language of some kind before introducing whatever conventional language is then fashionable or with a conventional language before introducing an array language or some applications package. There have been two very good examples of this in the Department of Computing Science. The first occurred in the mid 1960s when there was what might be politely called a "lively discussion" about using as a first language the machine-and-assembly-language package MENTOR/MENTORSAP which had been developed in the Department. This was eventually resolved amicably, and in a manner which is fortunately forgotten, over lunch in the Faculty Club. Then about thirty-five years later a similar situation arose when students in one course were first taught Pascal so that they would understand "programming" before being introduced to MATLAB. To me all arguments that a lower-level language must be learned before the introduction of a language which greatly simplifies the programming and solution of problems are fatuous. They deserve the response of the eccentric and reclusive English engineer and physicist Oliver Heaviside (1850 – 1925) who when hearing that a paper he had submitted to the *Proceedings of the Royal Society* had been rejected partly because of his notation said " Shall I refuse my dinner because I do not fully understand the process of digestion?"

Suppose now that we assume that a conventional language is introduced first, and leave further mention of array languages until the next section. There is just so much that may be reasonably introduced in a first computing course. Regardless of the language, a suggested list of topics, not necessarily in the order of presentation, is the following:

- Constants and variables.
- Scalars, and one- and possibly two-dimensional arrays.
- Operators.
- Assignment statements.
- Repetition (`for` and `while`).
- Selection (`if-then` and `if-then-else`).

- Input and output.
- Procedures and functions.

These concepts may be illustrated in lectures with simple examples and reinforced with simple assignments in which programming style and documentation are as important as getting the correct answer. Programming style should include the use of meaningful and suggestive names for variables and procedures with names such as `xxx` and the all-too-common `foo` being rigorously eschewed. We are in good company when we take care with the names we use for as one of the biographers of Charles Dickens remarks "Names were very important to Dickens. ... They [his characters] did not exist until he had given them a name and it is that which, like a spell, brings forth their appearance and behaviour in the world."

Any of the conventional languages introduced in this paper could be used to illustrate the programming concepts given above. However Fortran may have become mostly a legacy language useful for maintaining large projects without the need of reprogramming in a more recently introduced language, while Algol W is probably no longer available. Both C and Java might be considered too complicated to use as a first language, with Java in particular requiring the initial acceptance on faith alone of the protocol demanded in even the simplest of programs. It is too soon to give an opinion on Perl, and we should look forward with considerable interest to the experience with the proposed experimental course. This leaves BASIC and Pascal with the academic flavour and formal definition of Pascal probably making it the more respectable language for use in a university environment. However in my opinion BASIC with its historical roots and almost universal acceptance should not be ruled out.

The content of a first course

The discussion of the last section nicely avoids the question of how we should teach a programming language. Should we teach a programming language in a course whose primary intent is to teach the language with examples and applications chosen to illustrate the features of the langauge? Or do we introduce a programming language as required in the exposition of some topic such as computer logic, theory of computation, descriptive statistics or recreational mathematics? We shall consider these two alternatives briefly in this section and then suggest an alternative approach for an introductory course in computing.

Most introductions to computing appear to favour the first alternative. First programming courses are often introductions to syntax illustrated by a series of unrelated and occasionally silly examples. An example in one C++ text was a program that gave a prompt for the user to input a "favourite" number and gave the response `I think that [whatever number was input] is a nice number`. One Java text gave as an example a program to print either `ho-ho`, `he-he` or `ha-ha` which was then modified to print `yuk-yuk`. A programming assignment I saw in the Department many years ago was to prepare a table of n^7 and 7^n for a range of values of n, an exercise without interest or application. In my opinion it is difficult to justify the use of such material in a discipline which calls itself a science especially when it is part of a Faculty of Science.

About twenty-five years ago I coined the phrase "The Hello Fred School of Computing Science" after having seen too many assignments requiring the student to write a program which printed the text "Hello Fred" or a somewhat similar message. A short time ago I happened to see a test paper in the Department which in one question asked the student to write or to modify – I forget which – a program giving such a message. I suppose I can take comfort in learning that Fred is alive and presumably well after I have neglected him for so many years, but I was still disappointed in the reappearance of this example after an absence of so many years.

The second alternative of introducing a language in the treatment of some topic has never been popular in the Department. Some years ago Bill Adams tried this approach with Algol W, using it in a treatment of computer architecture. Also I used it one year with APL in a course in probability and statistics. Unfortunately none of our colleagues appeared to take any notice of our efforts. It is with such an approach that an array language – MATLAB, APL, **J,** or MINITAB or Mathematica which have never been used in the Department - is especially useful since much of the detail required in other languages may be suppressed and the emphasis placed on the subject matter.

We might note that Ken Iverson illustrated this approach in the small technical report *APL in Exposition*. In it he briefly showed how APL could be used in the teaching of various topics such as elementary algebra, coordinate geometry, finite differences, logic, sets, electrical circuits, and the design of a simple computer. The last section in just fifteen pages presented the logic of a simple computer for executing algebraic expressions, the parsing and compilation of compound expressions, and a program to compute a sequence of Fibonacci numbers.

The introduction of a programming language in some real situation is the way we learn our mother tongue when we are children, i.e., by using it to talk about the world around us. This world consists of our parents, and our brothers and sisters and aunts and uncles, and kittens and puppies, and trees and flowers, and ... . We gradually learn words and how to combine them first into simple phrases and then into sentences. Teaching grammar can wait until the child is in school and has some oral fluency. Also when we study a foreign language in school the texts often introduce the language in terms of a fictional although realistic story with the necessary vocabulary and grammar together with related exercises being introduced as the story unfolds.

Most introductory computing courses appear to be designed for students who are considering a career in computing or require some knowledge of computing in their own subjects. However such an approach ignores, however unintentionally, the very large majority of students who would be interested in some appreciation of the subject of computing as part of a liberal education without any regard for its immediate or long-term practical usefulness. This approach to teaching a science has been termed by one writer as treating “science as a humanity”. Some years ago I developed such a course for liberal arts students in which I took an historical approach to computing which I summarized somewhat casually as “beginning with ancient Egypt and ending with the current issues of *Scientific American*”. Some of the topics covered were number systems of past civilizations, the Hindu-Arabic number system and methods of performing arithmetic, logarithms, arithmetical and logical machines, Alan Turing and computability, the development of the electronic computer, and the evolution of programming languages. The APL language with its subordination of detail proved to be an effective means of presenting the computational aspects of topics discussed. In my opinion the course proved to be very popular with many students during the several years when it was given. Unfortunately in the mid 1980s it became a casualty of the computer literacy movement and has never been resurrected.

Let us end this section on teaching and learning a language, whether a natural language or a programming language, by mentioning *Richard Scarry’s Storybook Dictionary* (Paul Hamlyn, London, 1967) intended to introduce English to young children. It is a book which delights the adult reader as much as it does the child to whom it is being read. It is a large format book in which 2500 words are introduced by means of 1000 pictures through the adventures of such colourful characters as Ali Cat, Dingo Dog, Gogo Goat, Hannibal Elephant and Andy Anteater. In the Introduction we are told that “He [Presumably girls are included too.] will *not* be given rules. Rather, he will be shown by examples in contexts which completely catch his interest and hold his attention. ...” If we would only teach programming in the same way!

Acknowledgements

I would like to thank the following persons whom I consulted during the writing of this paper: Roger Hui for his suggestions on implementing binary arithmetic in **J**, Maria Stepanova for debugging the Fortran II program and for making me recall that Watfiv was not compatible with current versions of Fortran, Roman Fedoriw for debugging the C and Perl programs, Laura Watson for debugging my first (and I hope last) Java program, and Catherine Descheneau who read a draft of the paper. It was a pleasure working with these people, and their assistance is much appreciated.

References

Ackroyd, Peter. *Dickens*. Mandarin Paperbacks, London, 1994.

Adams, William S. *LGP-30 Lectures*. Unpublished lecture notes, 1960.

Bergin, Thomas J. and Richard G. Gibson (ed.). *History of Programming Languages-II*. Addison-Wesley Publishing Company, Reading, Mass., 1996.

Boswell, Clay S. Jr. *LGP-30 Act I Compiler*. Royal McBee Corporation, Port Chester, N. Y., 1959

"A brief history of programming languages," *BYTE.* September 1995. http://www.byte.com/art/9509/sec7/art19.htm.

Chapters from the Programmers' Handbook (Edition 2) for the Manchester Electronic Computer (Mark II). University of Toronto, Toronto, Ontario, 1953.

Charlton, Bruce. "When science should be a humanity," *New Scientist*, May 25, 1991, pp. 54 – 55.

Frankel, Stanley, "Useful applications of a magnetic-drum computer," *Electrical Engineering*, vol. 75, July, 1956, pp. 654 - 659.

Frankel, Stanley P., "The logical design of a simple general purpose computer," *IRE Transactions on Electronic Computers,* March, 1957, pp. 5 - 14.

Frankel, Stanley and James Cass, "LGP-30 General-Purpose Digital Computer," *Instruments & Automation*, vol. 29, Feb., 1956, pp. 264 - 270.

Gotlieb, C. C. and J. P. N. Hume. *High-Speed Data Processing*. McGraw-Hill Book, Inc., New York, 1958.

Iverson, K. E. *APL in Exposition*, IBM Philadelphia Scientific Center Technical Report No. 320-3010, 1972.

Keeping, E. S. *A Short History of the Department of Mathematics*. Department of Mathematics, University of Alberta, Edmonton, Alberta, 1971.

Kenney, J. F. and E. S. Keeping. *Mathematics of Statistics. Part One. Third Edition*. D. Van Nostrand Company, Inc., Princeton, N. J., 1954.

Larmour, Judy. *Laying Down the Lines: A History of Land Surveying in Alberta*. Brindle and Glass Publishing, Ltd., Edmonton, Alberta, 2005.

LGP-30 Programming Class Notes. Royal McBee Corporation, Port Chester, N. Y.

LGP-30 Programming Manual. Royal McBee Corporation, Port Chester, N. Y., 1957.

Pioneers in Computing. IBM Center for Advanced Studies, Toronto, Ontario, Oct. 19, 2005.

Nahin, Paul J. "Oliver Heaviside," *Scientific American*, June 1990, pp. 122 – 129.

Sammet, Jean E. *Programming Languages. History and Fundamentals*. Prentice-Hall, Inc., Engelwood Cliffs, N. J., 1969.

Smillie, Keith. "An historical approach to computing science,' *Historia Mathematica*, vol. 6, no. 1, 1979, pp. 63 – 66.

Smillie, Keith. *Programming Notes: APL, Fortran, Algol, Pascal*. Department of Computing Science, University of Alberta, Edmonton, Alberta, 1983.

Smillie, Keith. *The Department of Computing Science: The First Twenty-Five Years*. Technical Report TR 91-01, Department of Computing Science, University of Alberta, Edmonton, Alberta, 1991.

Smillie, Keith. *Computing Science at the University of Alberta 1957-1993.* Department of Computing Science, University of Alberta, Edmonton, Alberta, 1993.

Smillie, Keith. "From Abacus to Silion Chip," *http://www.cs.ualberta.ca/~smillie/Calculators/Calculators.html*, 2002.

Smillie, Keith. "Kenneth E. Iverson," *IEEE Annals of the History of Computing*, vol. 27, no. 4, pp. 93 – 96, 2005.

Thelen, Ed., *LGP-30*. *http://ed-thelen.org/comp-hist/lgp-30.html*, 2003.

Weik, M., *A Second Survey of Domestic Electronic Digital Computing Systems*, Office of Technical Services, United States Department of Commerce, Washington, D. C., 1957.

Wexelblat, Richard L. (ed.). *History of Programming Languages*. Academic Press, New York, 1981.

Williams, Michael R., "UTEC and Ferut: The University of Toronto's Computation Centre," *IEEE Annals of the History of Computing*, vol. 16, no. 2, 1994, pp. 4 - 12.

Keith Smillie is Professor Emeritus of Computing Science at the University of Alberta, Edmonton, Alberta T6G 2E8. His email address is smillie@cs.ualberta.ca.

Appendix. Programming examples

```
0000  b0125    Store number
0001  c0100        of faces
0002  b0039    Restore clear
0003  c0005        instruction
0004  c0035
0005  c0101    Clear freq. accumulator
0006  b0005    Increment clear
0007  a0037        instruction
0008  h0005
0009  s0040    More accumulators
0010  t0004        to clear?
0011  b0041    Restore pick-up
0012  c0013        instruction
0013  b0126    Pick up datum
0014  c0035    Is  face number
0015  s0035        positive?
0016  t0018    If +ve, continue
0017  z0000    Stop
0018  b0035    Shift face number
0019  m0038        and store
0020  c0036
0021  b0042    Set pick-up instruction
0022  a0036        for accumulator
0023  c0027
0024  b0043    Set put-away instruction
0025  a0036        for accumulator
0026  c0029
0027  b0100    Adjust total
0028  a0037        for face number
0029  c0100
0030  b0013    Increment pick-up
0031  a0037        instruction
0032  c0013        for face number
0033  u0013    Go to pick-up instruction
0034
0035           Store for number of faces
0036           Number of faces shifted
0037  z0001    Increment for instruction
0038  z0032    Increment for accumulators
0039  c0101    Restorer for clearing accumulators
0040  c0121    Test for end of clearing accumulators
0041  b0126    Restorer for pick-up instruction
0042  b0100    Restorer for accumulator pick-up
0043  c0100    Restorer for accumulator put-away
```

LGP-30 machine language

```
   LGP-30 ACT I Program for dice frequencies
   dim freq 20
   index i d
   read n
   1 : i
   s1 0 : freq i
   iter i 1 n s1
   s2 read d
   when d equal 0 trn s3
   freq d + 1 : freq d
   use s2
   s3 1 : i
   s4 0 print i 0 print freq i cr
   iter i 1 n s4
   stop
```

```
C     FORTRAN II PROGRAM FOR DICE FREQUENCIES
      DIMENSION IFREQ(20)
  100 FORMAT(5X,2I5)
      READ(5,100) N
      DO 1 I=1,N
      IFREQ(I)=0
    1 CONTINUE
    4 READ(5,100) NUM
      IF(NUM) 2,2,3
    3 IFREQ(NUM)=IFREQ(NUM)+1
      GO TO 4
    2 DO 5 I=1,N
      WRITE(6,100) I,IFREQ(I)
    5 CONTINUE
      STOP
      END
```

```
C     WATFIV FORTRAN PROGRAM FOR DICE FREQUENCIES
      INTEGER FREQ(20),D
  100 FORMAT(5X,2I5)
      READ(5,100) N
      DO 1 I=1,N
      FREQ(I)=0
    1 CONTINUE
      READ(5,100) D
      WHILE(D .GT. 0) DO
         FREQ(D)=FREQ(D)+1
         READ(5,100) D
      END WHILE
      DO 2 I=1,N
      WRITE(6,100) I,FREQ(I)
    2 CONTINUE
      STOP
      END
```

```
100 REM FIRST BASIC PROGRAM FOR DICE FREQUENCIES
110 DIM FREQ(20)
120 DATA 4,1,4,2,3,1,1,3,3,4,2,3,4,0
130 READ N
140 FOR I = 1 TO N
150    FREQ(I) = 0
160 NEXT I
170 READ D
180 IF D = 0 THEN 210
190 FREQ(D) = FREQ(D) + 1
200 GOTO 170
210 FOR I = 1 TO N
220    PRINT I, FREQ(I)
230 NEXT I
240 STOP
250 END
```

```
REM SECOND BASIC PROGRAM FOR DICE FREQUENCIES
DIM FREQ(20)
DATA 4,1,4,2,3,1,1,3,3,4,2,3,4,0
READ N
FOR I = 1 TO N
   FREQ(I) = 0
NEXT I
READ D
WHILE D > 0
   FREQ(D) = FREQ(D) + 1
   READ D
WEND
FOR I = 1 TO N
   PRINT I, FREQ(I)
NEXT I
STOP
END
```

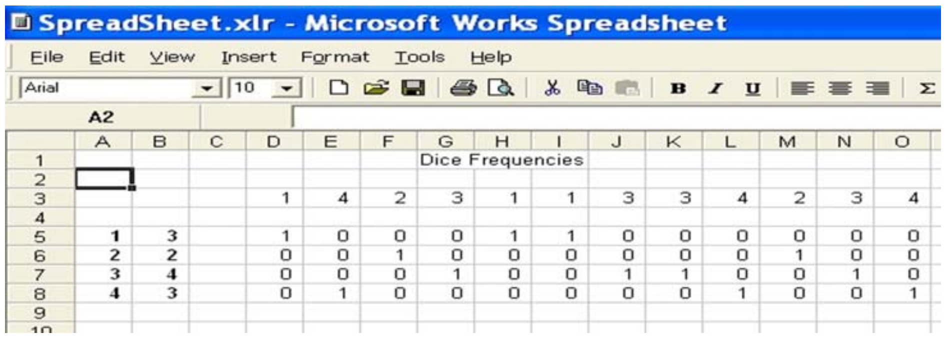
```
COMMENT
   ALGOL W PROGRAM FOR DICE FREQUENCIES;
BEGIN
INTEGER ARRAY FREQ(1::20);
INTEGER D,N;
READ(N);
FOR I:=1 UNTIL N DO FREQ(I):=0;
READON(D);
WHILE D > 0 DO
   BEGIN
   FREQ(D):=FREQ(D) + 1;
   READON(D);
   END;
FOR I:=1 UNTIL N DO WRITE(I,FREQ(I));
END.
```

```
(* Pascal program for dice frequencies *)
program DiceFrequencies(input,output);
type a = array[1..20] of integer;
var i,d,n  : integer;
    freq   : a;
begin
read(n);
for i:= 1 to n do freq[i]:= 0;
read(d);
while d > 0 do
   begin
   freq[d]:= freq[d] + 1;
   read(d);
   end;
for i:= 1 to n do writeln(i,'  ',freq[i]);
end.

main() /* C program for dice frequencies */
  {
     int d,i,n;
     int freq[21];
     scanf("%d",&n);
     for (i = 1; i < n; ++i)
        freq[i] = 0;
     scanf("%d",&d);
     while (d > 0) {
        ++freq[d];
        scanf("%d",&d);
     }
     for (i = 1; i <= n; ++i)
        printf("  %d   %d\n",i,freq[i]);
   }

// Java program for dice frequencies
public class DiceFrequencies {
   public static void main (String arg[]) {
   int d, i;
   int freq[] = new int [21];
   int D[] = {4,1,4,2,3,1,1,3,3,4,2,3,4,0};
   int N = D[0];
   for (i = 1; i <= N; ++i)
      freq[i] = 0;
   i = 1;
   while (D[i] > 0) {
      d = D[i];
      freq[d] = ++freq[d];
      i = ++i;
      }
   for (i = 1; i <= N; ++i)
      System.out.println (i + " " + freq[i]);
   }
}
```

```
# Perl program for dice frequencies
@D = (4,1,4,2,3,1,1,3,3,4,2,3,4,0);
$N = shift@D ;
foreach (1..$N){
   $freq[$_] = 0;
}
$d=shift@D ;
while ($d > 0) {
   $freq[$d] +=1;
   $d=shift@D ;
}
foreach (1..$N){
   printf("%5d %5d \n", $_ , $freq[$_] );
```


SpreadSheet.xlr - Microsoft Works Spreadsheet
File Edit View Insert Format Tools Help
Arial
10
A2

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1							Dice Frequencies								
2															
3				1	4	2	3	1	1	3	3	4	2	3	4
4															
5	1	3		1	0	0	0	1	1	0	0	0	0	0	0
6	2	2		0	0	1	0	0	0	0	0	0	1	0	0
7	3	4		0	0	0	1	0	0	1	1	0	0	1	0
8	4	3		0	1	0	0	0	0	0	0	1	0	0	1
9															

```
% MATLAB program for dice frequencies
N = 4;
D = [1 4 2 3 1 1 3 3 4 2 3 4];
Face = [1:N]';
Freq = zeros(N,1);
for i = 1:N
   Freq(i) = sum(D == i);
end
disp([Face Freq])
```

```
NB. J program for dice frequencies
pos=: >:@i.
Freq=: (pos@[) ,. [: +/"1 pos@[ =/ ]
```

LGP-30, BASIC and J programs for book prices example

```
0000  b0050    Initialize
0001  h0031       minimum,
0002  c0032       maximum,
0003  c0033       sum
0004  b0027    Restore pick-up
0005  c0006       instruction
0006  b0000    Pick up price
0007  c0030
0008  s0030
0009  t0011    Is price > 0?
0010  z0000    Stop
0011  a0031
0012  t0015    Is price > minimum?
0013  b0030    Set new minimum
0014  c0031       price
0015  b0030
0016  s0032
0017  t0020    Is price < maximum?
0018  b0030    Set new maximum
0019  c0032       price
0020  b0030    Add price to
0021  a0033       total
0022  c0033
0023  b0006    Increment pick-up
0024  a0028       instruction
0025  c0006
0026  u0006    Go to pick-up instruction
0027  b0050    Restorer for pick-up instruction
0028  z0001    Increment for instruction
```

```
REM Minimum, maximum and sum
DATA 20.00,10.99,23.95,25.95,24.00,23.95,0
Total = 0
READ Price
Minimum = Price
Maximum = Price
WHILE Price > 0
   IF Price < Minimum THEN Minimum = Price
   IF Price > Maximum THEN Maximum = Price
   Total = Total + Price
   READ Price
WEND
PRINT Minimum, Maximum, Total
STOP
END
```

```
   Prices=: 20.00 10.99 23.95 25.95 24.00 23.95
   Summary=: (<./,>./,+/)
   Summary Prices
10.99 25.95 128.84
```