Cabbage for Babbage's Analytical Engine

Andrew Davison

Dept. of Computer Eng., Prince of Songkla Univ.
Hat Yai, Songkhla 90110, Thailand

Email: ad@fivedots.coe.psu.ac.th

March 15th 2018

Abstract

The Cabbage compiler (http://fivedots.coe.psu.ac.th/~ad/cabbage/) translates a BASIC-like language into the instruction set created by John Walker for his emulator (https://www.fourmilab.ch/babbage/contents.html) of Charles Babbage's Analytical Engine [Bromley 1982]. The Cabbage language was developed in two stages: "raw" Cabbage supports only those high-level programming features that can be implemented on top of the existing engine, which include assignment, expressions, and restricted if and while statements. This illustrates in a practical way that Babbage's engine design is equivalent to a modern computer, not just an enhanced calculator (i.e. an improved difference engine). The second stage of the language, termed "boiled" Cabbage, adds arrays and subroutines which necessitates adding five new instructions to the engine. These allow a wide range of programs to be coded, including a version of the Countess of Lovelace's Bernoulli numbers algorithm [Kim and Toole 1999]. They also offer an opportunity to talk about the early design of hardware and programming languages.

1. Introduction

Cabbage is a BASIC-like language developed for an undergraduate course on compiler design. Initially I intended the compiler to generate byte codes for the JVM, but then I came across John Walker's Analytical Engine Simulator (https://www.fourmilab.ch/babbage/contents.html). The engine is simpler than the JVM (an important consideration for teaching), and also gives me a chance to talk about the early history of computingduring the subject. This led to Cabbage being designed in "raw" and "boiled" versions. "Raw" Cabbage supports only those high-level features that can be compiled down to existing engine instructions, while "boiled" Cabbage adds arrays and subroutines which, rather surprisingly (for students), only requires a handful of new instructions to be added to the engine.

Although Charles Babbage never built an analytical engine, he left descriptions spanning some 30 years [Bromley 1987]. Perhaps the most famous is the article written by the Countess of Lovelace [Lovelace 1843] which translates and expands upon an 1842 report of the engine by the Italian L. F. Menabrea.

Walker describes his design choices very clearly on his website (see "Is the Emulator Authentic? at https://www.fourmilab.ch/babbage/authentic.html), and his engine's instruction set are discussed on a page called "Programming Cards" (https://www.fourmilab.ch/babbage/cards.html).

Walker's emulator was too complex for my needs, so I stripped away all but the core functionality, leaving four classes:

1. AES.java: the top level of the emulator;
2. CardReader.java: it reads in a file, storing the instructions (the 'cards') in a list;
3. Mill.java: the equivalent of an ALU, but only capable of addition, subtraction, multiplication, and division, and the left and right shifting of numbers;
4. Store.java: The emulator's memory, made up of a 'rack' of 1000 'columns'. Each column is a memory cell that can hold a 50-digit integer.

The features removed include: all the Attendant request cards, the curve drawing cards, and GUI and Web/applet support. The attendant is a sysops person who manages the cards while they are being fed through the engine. His/her job includes switching on (and off) program tracing, formatting output, replacing control logic cards by simpler combinatorial cards, inserting library functions cards into the user's deck, and converting decimal input into integer form, and integer output back into decimals. These features aren't needed in my simplified engine because they're (mostly) handled by the Cabbage compiler.

What remains:

- Number cards for data;
- Operation cards for arithmetic (+, -, *, /);
- Variable cards for transferring data between the store and mill;
- Stepping Up and Down cards for shifting decimal data left and right (in an analogous way to the bitwise shift operators in C);
- Combinatorial cards for moving forwards and backwards through the cards in the reader;
- Action cards for ringing the bell, halting execution, and printing the last mill result;
- Comment cards

I won't describe these cards in detail since Walker's "Programming Cards" page (https://www.fourmilab.ch/babbage/cards.html) does a fine job of that. Instead, I'll explain two programs that use most of the cards.

1.1. Two Examples

The following code divides 10000 by 28 and prints the quotient and remainder:

```
N001 10000
N002 28

/
L001
L002
S003'
P
S004
P
H
```

The numbers are stored at 'columns' (locations) 1 and 2 by N (Number card) operations, then the mill is initialized with the division operation, and the numbers loaded into it with two L instructions.

The mill has two output registers called the Egress Axis and Primed Egress Axis; after a division, the primed egress axis holds the quotient, which is stored to location 3 (the ' indicates the use of the primed axis) and printed. The remainder is in the egress axis, which is copied to location 4 and also printed. The output is:

```
357
4
```

The next example calculates the factorial of 6 using a conditional loop:

```
N0 6   . start value, decreases by step
N1 1   . result
N2 1   . step
*
L1
L0
S1
-
L0
L2
S0
-
L2
L0
CB?12

*     . done so result is loaded into mill for printing
L1
L2
S1
P
H
```

The text after the "."s are comments. The loop is implemented using the combinatorial card "CB" which jumps back 12 cards when the engine's run up lever is set. The card count includes the CB card, so the jump will land on the earlier * card.

The run up lever can be set for a number of reasons including when the mill has just performed a subtraction and a change of sign is detected. In the example, the subtraction before the CB card is (- L2 L0). If the result is negative **and** L2 is positive then the run up lever is set and a jump will occur from the CB card.

L2 loads the step data in location 2 into the mill, while L0 loads the start value from location 0, and this value is decremented by 1 on each loop. While the value is 6, 5, 4, etc., the subtraction prior to the CB card will be negative and a jump will occur. But when the value reaches 1, the subtraction returns 0, and the run up lever is **not** set. At that point, execution moves forward, leaving the loop.

1.2. The Importance of the Combinatorial Cards

The combinatorial cards are essential for implementing conditional jumps, which means that the Analytical Engine is a Turing machine [Turing 1936], and so mathematically equivalent to modern computer hardware.

Babbage never built an engine, so it is interesting to consider whether the functionality offered by the combinatorial cards is actually possible. For example, 20th Century punch card readers could not backup through a deck.

Babbage envisaged the cards being tied together in a looped chain (as they were in Jacquard's looms), so backward movement could be implemented by a forward move around the loop back to the beginning of the cards. This would necessitate the reader 'knowing' the length of the deck so it could calculate the number of moves. A similar approach was used in the 1950's but in paper tape readers.

An interesting consequence of this mechanical approach is that it isn't necessary for the deck's code to be stored in memory in order for execution to utilize loops and branches. In other words, stored program hardware isn't a prerequisite for a Turing machine, so long as the reader can move forwards and backwards over the program. This, of course, was one of the assumptions in Turing's model. The practical downside of this approach is that looping through a chain of cards in this way would probably take minutes. Indeed, access speed was one of the reasons for the introduction of the stored program idea [von Neumann 1945].

2. Raw Cabbage

The BNF for the "raw" version of Cabbage is:

```
<program> ::=  [ <numdps> ] <stat> { <stat> }
<numdps> ::= "numdps" "=" <number> ";"
<stat> ::=  <assign> ";"
               | if "(" <cond> ")" <stat> [ else <stat> ]
               | "{" <stat> { <stat> } "}"
               | while "(" <cond> ")" <stat>
               | print <expr> ";"
               | ε
<assign> ::=  <var> "=" <expr>
               |   "(" <var> "," <var> ")"  "=" <argum> "/" <argum>
<cond> ::= <expr> ">=" <expr>
<expr> ::= [ ("+" | "-") ] <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::=  <argum> |  "(" <expr> ")"
<argum> ::= <var> | <number>
<var> ::= <ident>
```

Two somewhat non-standard aspects of this definition are the special assignment for division, and the limited conditional (which only offers ">=").

The two programs given earlier can be recoded. The division program becomes:

```
x = 10000;
y = 28;
(q, r) = x / y;
print q;
print r;
```

Note: if you try this is the final ("boiled") version of Cabbage, then you must also define at the top of the program a value for the built-in numdps variable:

```
numdps = 0;
```

I'll explain what numdps does in the next section.

Factorial is coded a little differently from the engine version, due to the restrictions on the while loop's conditional:

```
// 6! example
n = 6;
result = 1;
step = 1;
while (n >= step) {
  result = result * step;
  step = step + 1;
}
print result;
```

The only conditional form is x >= y, which is translated into a subtraction at the engine level (- x y). As before, the run up lever will be set when the subtraction result is negative, and the left-hand operand (i.e. x) is positive. This means that a Cabbage programmer must ensure that the expression on the left of the ">=" is positive or the run up lever will not be set.

2.1. From Integer to Fixed-Point Arithmetic

The engine supports 50-digit integers only, but Babbage was well aware that decimal point calculations were necessary. His solution was to have the engine's attendant translate numbers with fractional parts into integers before passing them to the engine by multiplying through by a sufficient power-of-10. For example, the equation:

(4000 * 2.5) / 2.8

could have its numbers multiplied by 10^6:

(4000000000 * 2500000) / 2800000

Instead of 3571 the result is 3571428571, which can be reported by the attendant as 3571.428571, a result with 6 decimal places.

Unfortunately, the attendant will have to modify almost every multiplication and division so the result stays scaled by the correct power-of-10. For example, each multiplication will need an excess 10^6 removed from the answer, and a divisor will need another 10^6 multiplied to it so the result is scaled correctly.

For example, 1/3 as 1000000/3000000 would still produce the integer 0; its divisor needs to multiplied by another 10^6 so the result is scaled in the same way as the two inputs, as 333333. The attendant reports this as 0.333333.

The good news is that Cabbage does this scaling automatically by judiciously inserting calls to the engine's stepping up and down (shift) instructions. By default, Cabbage assumes that 8 decimal places are required, so all numbers are stored after being multiplied by 10^8. The number of decimal places can be adjusted by including a `numdps` statement at the start of the program. For example:

```
numdps = 0;
```

switches off scaling.

Since "raw" Cabbage adds nothing to the engine's instruction set, it's not possible to print numbers with a correctly placed decimal point; only as integers.

2.2. Dealing With Expressions

The BNF sports a familiar set of rules for defining an expression using terms and factors which define the precedence for the arithmetic operations:

```
<expr> ::= [ ("+" | "-") ] <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::=  <argum> |  "(" <expr> ")"
```

Most virtual machine's evaluate an expression using a stack, pushing the operator and two operands onto it before popping the three elements to calculate the result. One advantage is that an operand can be a sub-expression, and the stack 'remembers' the enclosing expression while the sub-part is evaluated.

Unfortunately, the engine hasn't got a stack. The only solution which doesn't extend the engine's instruction set is to have the compiler rewrite expressions as multiple assignments. For example:

```
result = (4000 * 2.5) / (28 - 3);
```

is treated as:

```
temp1 = 4000 * 2.5;
temp2 =  28 - 3;
result = temp1 / temp2;
```

The drawback is the additional variables (e.g. temp1 and temp2), which may become a problem since the store only has space for 1000 variables.

2.3. Limits on Assignment

The S card (for storing a value in a variable) assumes that the value is in one of the mill's output registers. This means that a high-level assignment such as:

```
x = result;
```

cannot be encoded directly at the engine level since the value on the right-hand side must be generated by a mill calculation. The solution is to perform a dummy calculation equivalent to:

x = result + 0;

The Cabbage compiler adds these extra calculations as necessary.

3. Boiled Cabbage

The "boiled" version of Cabbage adds arrays and functions to the language.

As mentioned above, my intention is to use Cabbage and the simplified emulator in a compiler construction subject, and so I wanted to make the smallest number of changes to the engine's instruction set as possible when extending Cabbage.

It turns out that array support requires the engine's load and store instructions (L, Z, and S) to be joined by indirect versions (called "l", "z", and "s"). Functions needs two additional instructions ("v" and "J") for accessing the program counter and performing a jump using a distance computed at run time.

I also added some basic IO features, and the ability to call Java functions. My aim with the latter was to allow the use of functions external to the engine, such as those in Java's Math class so I didn't have to re-implement every function in Cabbage.

The BNF for "boiled" Cabbage is given below. The new and changed rules are highlighted in bold:

```
<program> ::=  [ <numdps> ] <main> { <func> }
<numdps> ::= "numdps" "=" <number> ";"

<main> ::= <stat> { <stat> }
<func> ::= "func" <ident> "(" [ <ident> { "," <ident> } ] ")"
           "{" <stat> { <stat> } "}"

<stat> ::=  <assign> ";"
              | if "(" <cond> ")" <stat> [ else <stat> ]
              | "{" <stat> { <stat> } "}"
              | while "(" <cond> ")" <stat>
              | read <var> ";"
              | print ( <expr> | <string> )
                    { "," ( <expr> | <string> ) } ";"
              | dim <ident> "[ <number> "]" ";"
              | <call> ";"
              | return <expr> ";"
              | ε

<call> ::= ( "call" | "ext" )
               <ident> "(" [ <expr> { "," <expr> } ] ")"

<assign> ::=  <var> "=" <call> | <expr>
              |   "(" <var> "," <var> ")"  "=" <argum> "/" <argum>

<cond> ::= <expr> ">=" <expr>
<expr> ::= [ ("+" | "-") ] <term> { ("+" | "-") <term> }
```

```
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::=  <argum> |  "(" <expr> ")"
<argum> ::= <var> | <number>

<var> ::= <ident> [ "[" ( <ident> | <number> ) "]" ]
```

The syntax is standard for a BASIC-like language, although simplified. For example, an array index cannot be an expression (e.g. A[i+2] is not allowed), and a function cannot be called inside an expression.

3.1. Adding Arrays

Having a way to store a sequence of data is a natural language extension for two reasons. First, sequences are a common occurrence in mathematics, based around subscripted variables, so adding them to Cabbage would be useful for math programming. Secondly, once a language has looping capabilities, then it becomes useful to iterate over a sequential data structure. It should be noted that I've been talking about sequences rather than arrays, since lists would fit the bill just as easily.

Array manipulation was seen as necessary from the start of modern computing – it is present in John von Neumann's Draft report on the EDVAC [von Neumann. 1945], but implemented using instruction modification. This approach was utilized for a range of features including conditional branching and subroutines, and was another reason for employing the "stored program" concept – if the code is in memory then it's easier to change.

However, index registers and indirect addressing eventually won out due to their simplicity. The first index register mechanism was the B-line in the Manchester Mark 1 (so-named because the letters A, and C had already been used) [Lavington, 1980]. The B-line utilized the Willliams-Kilburn tube, a cathode-ray device, as random-access memory. It was lighter, cheaper, and offered more flexible access than J. Presper Eckert's mercury delay line utilized in the EDVAC.

The fundamental change for the engine is the realization that variables are able to store two kinds of data: numerical values (as before) and addresses of other variables. This distinction wasn't needed in the original engine since combinatorial cards make jumps relative to the current card.

The engine's indirect addressing relies on the compiler representing an array by *two* variables in the store. The first is a contiguous block of cells for the elements of the array (e.g. A[], starting at address 9), and the second is a variable that stores the start address of that array (e.g. an A_start variable is assigned 9). In a Cabbage program, a reference to an array (e.g. A[2]) is converted into an addition of 2 to A_start (e.g. 2 + 9). Indirect addressing means that the result (11) is treated as an address (the third cell of the A array), not a number.

3.2. Coding Bernoulli Numbers

An array capability makes it relatively easy to implement (and extend) Lovelace's Note G for generating the fourth Bernoulli number [Lovelace, 1843]. This was translated into engine instructions (with some errors corrected) by Simon Wright in 2012 as an example for his Ada version of Walker's emulator (https://github.com/simonjwright/analytical-engine/blob/master/bernouilli.ae). Lovelace's pseudo-code and Wright's program do not use loops since there's no way to iterate over subscripted variables. Instead the loop is essentially partially evaluated, to become a sequence. A good modern description of Lovelace's approach can be found at https://enigmaticcode.wordpress.com/tag/bernoulli-numbers/.

Generating Bernoulli numbers is a popular coding task, and is represented at the Rosetta Code website (https://rosettacode.org/wiki/Bernoulli_numbers) and the Code Golf section of StackExchange (https://codegolf.stackexchange.com/questions/65382/bernoulli-numbers/65393). The best approaches report their results as rationals rather than as rounded decimals, which requires something like the Apache Commons Maths BigFraction library in Java (http://commons.apache.org/proper/commons-math). The Cabbage solution uses 15dp fixed decimals, and generates all the non-zero values up to the 58th Bernoulli number:

```
numdps = 15;

maxBerns = 30;

dim a[61];     //  2*maxBerns+1
dim bs[61];

bs[1] = 1/6;

n = 2;
while (maxBerns >= (n+1)) {      // n < MAX_BERNS
  a[0] = 1/2 * (2*n-1)/(2*n+1);
  a[1] = n;

  i = 2;
  while ((2*n-3) >= i) {
    previ = i-1;
    a[i] = a[previ] * (2*n - (i-1))/(i+1);
    i = i+1;
  }

  sum = a[0];
  j = 3;
  while ((2*n-1) >= j) {
    prevj2 = j-2;
    sum = sum - (a[prevj2] * bs[prevj2]);
    j = j+2;
  }

  bsVal = 2*n-1;
  bs[bsVal] = sum;
  print "B(", 2*n, ") = ", bs[bsVal];
  n = n+1;
}
```

Two arrays are used: bs[] for the Bernoulli numbers and a[] for the coefficient in the equation:

```
a[0] + a[1]*bs[1] + a[3]*bs[3] + ... + a[2n-3]*bs[2n-3] + bs[2n-1] = 0
```

The values of the a[] coefficients are defined as follows:

```
    a[0] = (1/2)(2n – 1)/(2n + 1)
    a[1] = 2n/2 = n
    a[3] = 2n(2n - 1)(2n - 2) / (2.3.4)
    a[5] = 2n(2n - 1)(2n - 2)(2n - 3)(2n - 4) / (2.3.4.5.6)
    ...
```

For k > 1, each a[k] is derived from a[k − 2] by multiplying the next two decreasing terms into the numerator, and the next two increasing terms into the denominator. Note that the n refers to the nth Bernoulli number so is different for each number.

Given the already computed a[] values and bs[1], bs[3], …, bs[2n − 3], then the first equation can be used to obtain bs[2n-1].

The output from the program begins like so:

```
B(4) = -0.033333333333332
B(6) = 0.023809523809519
B(8) = -0.033333333333306
B(10) = 0.075757575757331
```

As rationals these are -1/30, 1/42, -1/30, and 5/66. Even with 15 decimal place accuracy, rounding errors start to accumulate, and the results become noticeably incorrect by around bs[26]. For example, the output is:

```
B(26) = 1425516.8684888291
```

The correct answer is 142551**7**.16666… (8553103/6).

The rounding problem with fixed point numbers was a major motivation for the development of floating point arithmetic in the 1950s, first in software and later in hardware. Floating point hardware actually predates the advent of electronic digital machines: it appeared to be fairly 'easy' to build into the early relay calculators of Konrad Zuse (1936), George Stibitz (1939), and Aiken's Mark II (1944). However, it wasn't utilized in the ENIAC (1946) since John Mauchly and J. Presper Eckert decided that floating point circuitry was too hard to implement with tubes, and consume too much memory since exponents had to be stored [Knuth 1997].

The first floating point subroutine libraries were coded by David Wheeler and others for the EDSAC, and were widely disseminated through the first programming textbook: "The Preparation of Programs for an Electronic Digital Computer" by Wilkes, Wheeler, and Gill [1951]. Floating decimal subroutines were described, although EDSAC was a binary computer.

Another widely discussed problem illustrated by my Bernoulli Cabbage code is the bloated translation from high-level notation to engine instructions. My compiled code weighs in at 448 lines compared to Wright's lean 274 lines, although his program only computes a single Bernoulli number. Until the mid 1950's high-level notations such as Glennie's AUTOCODE (1952) for the Manchester Mark I didn't catch on because being able to fine-tune machine code for space efficiency was deemed more important than high-level support for assignment, expressions, and conditional statements [Knuth and Trabb Pardo, 1977].

3.3. Fancier Output

The Bernoulli program includes an example of the Cabbage print statement:

```
print "B(", 2*n, ") = ", bs[bsVal];
```

The BNF for the print statement is:

```
print ( <expr> | <string> ) { "," ( <expr> | <string> ) } ";"
```

This requires the addition of two new instructions to the engine: "U" (putchar) and "D" (print a double).

The most important aspect of adding putchar is the conceptually extension of the engine's data types once again so that a number in memory can be an encoded character (in addition to a number and memory address).

Babbage was familiar with the notion of encoding text through the electric telegraph [Hyman 1985] which was being developed and installed widely in the UK throughout the 1840s and 1850s. William Cooke and Charles Wheatstone built a telegraph in 1841 that printed the letters from a wheel of typefaces struck by a hammer, and Babbage examined some of Wheatstone's work at around this time. One of the mysteries of Babbage's work on the engine was why he never considered utilizing electromechanical switching instead of solely mechanical techniques.

The "D" instruction is part of the support for fixed point decimals at the Cabbage level. The engine's original print instruction, "P", prints a the number in the mill's output register as an integer. The "D" instruction adds a "." before printing the decimal part.

3.4. Adding Subroutines

The utility of subroutines was well known to Babbage, who envisaged that the engine's attendant would insert the necessary cards into a user's deck before the program was run. In modern terminology, these would be termed "open subroutines".

"Closed subroutines" are functions that are linked to the user's program at run time. This linking requires that the called subroutine has access to a return address so that when it has finished its work, it can jump back to the correct place in the user's program. This return address must be supplied by the calling program which means that the program counter (the address of the code currently being executed) must be accessible.

This mechanism was described in von Neumann's draft report for the EDVAC [von Neumann 1945], but Maurice Wilkes' EDSAC [Campbell-Kelly 1980], completed and running in 1949, was the first machine to offer an implementation based around the Wheeler jump, named after David Wheeler [Wheeler 1952].

The first EDSAC subroutines were for division (there was no hardware divider in the machine), square root, and the transcendental functions. Later, subroutines for input and output and more complex mathematical procedures were added. By the end of 1950, the library contained some 80 subroutines, some contributed by users.

3.5. A Factorial Example

The following "boiled" Cabbage program employs a factorial function:

```
read n;
```

```
f = call fact(n);
print n, "! = ", f;

f = call fact(9);
print "9! = ", f;


func fact(n)
{
  prod = 1;
  i = 1;
  while (n >= i) {
    prod = prod * i;
    i = i + 1;
  }
  return prod;
}
```

The main program calls the function twice: once with the user's input as the parameter, once to calculate 9!. The output is:

```
  ? 5
5! = 120
9! = 362880
```

The "?" on the first line is generated by the read statement, and the user enters 5.

3.6. Implementing Subroutines

Calling a function is not only a matter of jumping to the start of its code; various bits of housekeeping must be performed as well. Firstly, the location of the function call in the program must be recorded, which necessitates access to the program counter. This is retrieved in the engine with the "v' instruction, and stored in a variable called "ret_fact" for the example above. Parameter passing must also be handled, which is kept simple by having the engine only support the passing of numbers, not arrays. This means that all data is passed call by value (i.e. by copying), so call by reference, or more esoteric mechanisms, aren’t needed.

The compiler knows how many arguments are used by a function, and so a series of new variables, called <function_name>_0, <function_name>_1, etc. are assigned the function's arguments. Completing the set of new variables is "result_<function_name>", which is used to hold a function's result so it can be accessed by the calling function.

These new variable names might clash with the names used by the programmer, but that's avoided by renaming the program variables. The variables inside a function are prepended with the function's name, and the variables in the main function (e.g. the first five lines of the factorial example) have "main_" added. This renaming offers a primitive form of lexical scoping because it makes it impossible for a function to employ a variable outside its scope. However, renaming isn't performed on array names so these could be used as global variables if necessary.

A function ends with another call to the "v" instruction to get the current address. The "ret_fact" value is subtracted from this address to get the jump distance back to the calling function, employed by the "J" instruction.

Back in main, the "result_fact" value (the result of the factorial call) is assigned to the "f" variable.

In summary, subroutine functionality can be supported in the engine by the addition of two new instructions: "v" to access the program counter, and "J" to jump based on the output of the mill.

3.7. Recursion

The drawback of this approach is that recursion isn't possible. The problem lies in the use of predefined variables to hold subroutine details – the input parameters, the result, and (most importantly) the return address. If the factorial function calls itself recursively, then each call would overwrite the information stored during the previous call.

The solution is to store function details on a stack: a new 'frame' of variables is created at call time and pushed onto the stack , then popped off when the function returns.

Perhaps surprisingly, a stack mechanism for subroutines (called "reversion storage") was described before the invention of the Wheeler jump, in the proposal by Alan Turing for his ACE computer [Turing 1946]. Amusingly, instead of push and pop he used the terms "bury" and "disinter/unbury". Only a much-simplified form of the ACE was finally completed in 1950, the Pilot ACE, which didn't include Turing's subroutine technique [Campbell-Kelly 1981].

Stanley Gill, one of the developers of the subroutine system in the EDSAC, worked under Turing for two years up to October 1949, and a call stack was added to the EDSAC in 1951 (or a bit later) [Brooker and Wheeler 1953].

Recursion was also slow to arrive as a high-level programming feature. Although FORTRAN II (1958) offered subroutines and functions (with parameters passed by reference), and the COMMON statement for global variables, it didn’t have recursion, mainly because popular IBM machine's of the time (e.g. the IBM 1130) had no stack support [Backus 1979]. Also, the way that recursion appeared in ALGOL-60, due to the 'machinations' of Adriaan van Wijngaarden and Edsger Dijkstra makes for an interesting historical read (https://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/).

3.8. External Functions

Another path to follow from this stage in Cabbage's design is the addition of external libraries, which would require the introduction of loading and linking techniques, and the notion of code with relocatable addresses. Once again, Wilkes' EDSAC was the first with this capability, in its "assembly routines" in 1950.

My solution for using external functions (i.e. ones not implemented in Cabbage), breaks from an historical approach and relies on Java's reflection mechanism (although you could argue that it's sort of similar to EDVACs code modification).

The following program uses an external random() function to implement a Monte Carlo way of calculating pi:

```
nThrows = 0;
nSuccess = 0;

i = 0;
while (100000 >= i) {
  x = ext random();
  y = ext random();
  nThrows = nThrows+1;
  if (1 >= x*x + y*y)
    nSuccess = nSuccess+1;
  i = i+1;
}

res = 4 * nSuccess / nThrows;
print "pi = ", res;
```

The Monte Carlo method was one of the first techniques used by von Neumann on the ENIAC [Goldstein 1972], following its modern-day reinvention by Stanislaw Ulam in 1947. Von Neumann got round his problem of generating random numbers by implementing a version utilizing the middle-square method.

In my program, a random (x, y) coordinate is generated in the top-right quadrant of the x-y axis, between 0 and 1. If the point is within 1 unit of the origin then it is recorded as a success. Over a sufficient number of runs, the fraction of successes to the total number of runs will draw close to pi/4.

The output of this program after 100,000 iterations is:

```
pi = 3.14360856
```

The random() function is defined in a Java Utils class which I added to the engine's classes. It holds static functions which can take up to four double arguments and must return a single double result. For example:

```
public class Utils
{
  public static double log(double x)
  {  return Math.log(x);  }

  public static double random()
  {  return Math.random();  }

  public static double time()
  {  return (double) LocalTime.now().toSecondOfDay();  }

  // more public static double functions...

}  // end of Utils class
```

Cabbage stores the details of a call to an "ext" function in seven predefined global variables located at the start of memory ("func_Nm", "func_nargs", "func_arg0", "func_arg1", "func_arg2", "func_arg3", and "func_result"). The external function can be passed at most four double arguments, with the exact number stored in "func_nargs". It returns a single double which is stored in "func_result".

At the engine level, an "X" instruction uses the data stored in the globals to construct a call to the Utils class.

4. Summary

I have described a simple imperative language called Cabbage which is compiled into instructions for a cut-down version of John Walker's simulator for Babbage's Analytical Engine. Cabbage was developed in two stages (known as "raw" and "boiled"): the "raw" version utilizes only the emulator's original instruction set, and supports assignment, expressions, and if and while statements; the "boiled" version adds arrays, functions, and simple I/O.

"Raw" Cabbage illustrates in a practical way that Babbage's engine was a Turing machine, and the "boiled" version shows that the addition of a handful of new instructions (three for arrays and two for subroutines) would have made the engine easily capable of supporting imperative languages similar to BASIC.

Running through my description of Cabbage are references to the history of programming, which show that the necessary elements for indexing, subroutines, and recursion were understood from the very early days of computing.

The source code for the Cabbage compiler and the simplified Analytical Engine emulator are available at http://fivedots.coe.psu.ac.th/~ad/cabbage/.

References

Backus, J. 1979. "The History of FORTRAN I, II, and III", Annals of the History of Computing, Vol. 1, No. 1, July

Brooker, R.A., Wheeler, D.J. 1953. "Floating Operations on the EDSAC", Mathematical Tables and Other Aids to Computation 7, pp.37-47

Bromley, A.G. 1982. "Charles Babbage's Analytical Engine, 1838", Annals of the History of Computing, Vol. 4. No. 3, pp. 196-217

Bromley, A.G. 1987. "The Evolution of Babbage's Calculating Engines", Annals of the History of Computing. Vol. 9. No. 2, pp. 113-136

Campbell-Kelly, M. 1980. Programming the EDSAC: Early Programming Activity at the University of Cambridge", Annals of the History of Computing, Vol. 2, No. 1, pp.7-36

Campbell-Kelly, M. 1981. "Programming the Pilot ACE: Early Programming Activity at the National Physics Laboratory", Annals of the History of Computing, Vol. 3, No. 2, pp. 133-162

Goldstine, H.H. 1972. *The Computer: From Pascal to von Neumann*, Princeton Univ. Press, Princeton

Hyman, A. 1985. *Charles Babbage: Pioneer of the Computer*, Princeton Univ. Press

Kim, E.E. and Toole, B.A. 1999. "Ada and the First Computer", Scientific American, May, pp.76-81.

Knuth, D.E. 1997. *The Art of Computer Programming: Semi-numerical Algorithms*, Addison-Wesley, Vol. 2, 3rd ed.

Knuth, D.E., Trabb Pardo, L. 1977. "The Early Development of Programming Languages", In: *Selected Papers on Computer Languages*, D.E. Knuth, pp.1-93, Stanford (2002).

Lavington, S. 1980. *Early British Computers*, Manchester Univ. Press

Lovelace, A.A. 1843. "Sketch of the Analytical Engine Invented by Charles Babbage, by L.F. Menabrea, Officer of the Military Engineers, with notes upon the Memoir by the Translator", Taylor's Scientific Memoirs, Vol. 3, Article 39, 1843, pp. 666-731. Online at http://www.fourmilab.ch/babbage/sketch.html, last accessed January 16th, 2018.

Turing, A.M., 1936. "On Computable Numbers, with an Application to the Entscheidungsproblem", Proc. London Mathematical Society (Series 2), Vol. 42, pp. 230-267. Online at https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf, last accessed Feb 10th 2018

Turing, A.M. 1946. "Proposal for Development in the Mathematics Department of an Automatic Computing Engine (ACE)", Technical Report E882, National Physical Laboratory, Teddington, UK, February.

von Neumann, J. 1945. "First Draft of a Report on the EDVAC", Moore School of Electrical Eng., Univ. of Pennsylvania, Philadelphia. Reprinted in IEEE Annals of the History of Computing, Vol. 15, No. 4, pp.27-75 (1993)

Wheeler, D.J. 1952. "The Use of Sub-routines in Programmes", Proc. of the 1952 ACM National Meeting, Pittsburgh, pp. 235-236

Wilkes, M.V., Wheeler, D.J., Gill, S. 1951. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley