CONFERENCE
MILAN 2022

Paolo Caressa

“Programming a Problem Oriented Language”: 50 years later

A commentary on Chuck Moore’s unpublished book

Why a commentary to a book?

IMHO some landmark books in the history of computer programming are

- D.E. Knuth, *The Art of Computer Programming* ($\geq$ 1968).
- C. Moore, *Programming a Problem Oriented Language* (1971)
- N. Wirth, *Algorithms + Data Structures = Programs* ($\geq$ 1976).
- H. Abelson, G.J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs* ($\geq$ 1984).
- B. Kernighan, R. Pike, *The Practice of Programming* ($\geq$ 1999).

Moore's book was never published in English (it had several machine language translations known as Forth systems), look for it here: *http://forth.org/POL.pdf*.

Stacks everywhere

Stacks were invented by Alan Turing in 1946 to program subroutines. In 1957 Klaus Samelson and Friedrich Bauer unaware of Turing contribution filed a patent for it.

In the 60s stack-based interpreter for the Algol-60 language emerged (E.W. Dijkstra) and in 1971 Niklaus Wirth programmed the first Pascal to compile a byte-code for a stack VM.

In 1971 Moore finished his book, containing ideas developed in the 60s and exerting a great influence in the 1970s/1980s.

From 1990s on, most languages used stack virtual machine as byte-code: Java, C#, Python etc.

Stacks everywhere

Stacks were invented by Alan Turing in 1946 to program subroutines. In 1957 Klaus Samelson and Friedrich Bauer unaware of Turing contribution filed a patent for it.

In the 60s stack-based interpreter for the Algol-60 language emerged (E.W. Dijkstra) and in 1971 Niklaus Wirth programmed the first Pascal to compile a byte-code for a stack VM.

In 1971 Moore finished his book, containing ideas developed in the 60s and exerting a great influence in the 1970s/1980s.

From 1990s on, most languages used stack virtual machine as byte-code: Java, C#, Python etc.

Stacks everywhere

Stacks were invented by Alan Turing in 1946 to program subroutines. In 1957 Klaus Samelson and Friedrich Bauer unaware of Turing contribution filed a patent for it.

In the 60s stack-based interpreter for the Algol-60 language emerged (E.W. Dijkstra) and in 1971 Niklaus Wirth programmed the first Pascal to compile a byte-code for a stack VM.

In 1971 Moore finished his book, containing ideas developed in the 60s and exerting a great influence in the 1970s/1980s.

From 1990s on, most languages used stack virtual machine as byte-code: Java, C#, Python etc.

Stacks everywhere

Stacks were invented by Alan Turing in 1946 to program subroutines. In 1957 Klaus Samelson and Friedrich Bauer unaware of Turing contribution filed a patent for it.

In the 60s stack-based interpreter for the Algol-60 language emerged (E.W. Dijkstra) and in 1971 Niklaus Wirth programmed the first Pascal to compile a byte-code for a stack VM.

In 1971 Moore finished his book, containing ideas developed in the 60s and exerting a great influence in the 1970s/1980s.

From 1990s on, most languages used stack virtual machine as byte-code: Java, C#, Python etc.

(Well known) BASIC Principles

Moore's techniques stems from some well known principles, that are corollaries of the basic one:

```
10 REM KEEP IT SIMPLE, STUPID

20 REM DO NOT SPECULATE
25 REM (LATER: YOU AREN'T GONNA NEED IT)

30 REM DO IT BY YOURSELF(!)

40 REM PREMATURE OPTIMIZATION IS THE ROOT
45 REM OF ALL EVIL (D.E. KNUTH)
```

(Well known) BASIC Principles

Moore's techniques stems from some well known principles, that are corollaries of the basic one:

```
10 REM KEEP IT SIMPLE, STUPID

20 REM DO NOT SPECULATE
25 REM (LATER: YOU AREN'T GONNA NEED IT)

30 REM DO IT BY YOURSELF(!)

40 REM PREMATURE OPTIMIZATION IS THE ROOT
45 REM OF ALL EVIL (D.E. KNUTH)
```

(Well known) BASIC Principles

Moore's techniques stems from some well known principles, that are corollaries of the basic one:

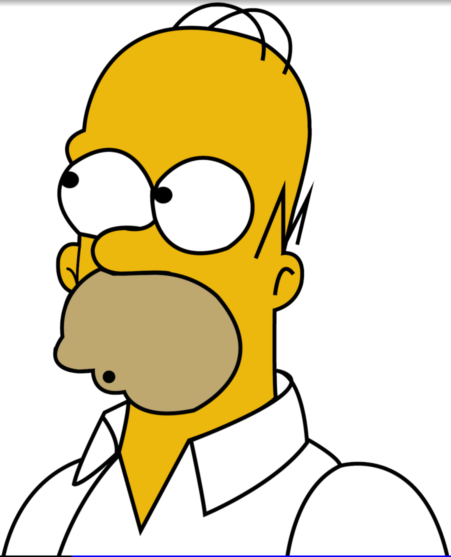
```
10 REM KEEP IT SIMPLE, STUPID

20 REM DO NOT SPECULATE
25 REM (LATER: YOU AREN'T GONNA NEED IT)

30 REM DO IT BY YOURSELF(!)

40 REM PREMATURE OPTIMIZATION IS THE ROOT
45 REM OF ALL EVIL (D.E. KNUTH)
```

(Well known) BASIC Principles

Moore's techniques stems from some well known principles, that are corollaries of the basic one:

```
10 REM KEEP IT SIMPLE, STUPID

20 REM DO NOT SPECULATE
25 REM (LATER: YOU AREN'T GONNA NEED IT)

30 REM DO IT BY YOURSELF(!)

40 REM PREMATURE OPTIMIZATION IS THE ROOT
45 REM OF ALL EVIL (D.E. KNUTH)
```

By means of these principles we'll implement a programming language!



Peano principle: avoid grammar

A source program is a text file, whom our interpreter will parse *words* to be executed/compiled: a word is a sequence of non blanks characters delimited by blanks (some characters count as a single word, more on that later).

$$\boxed{\text{PRINT}}\ \boxed{1}\ \boxed{+}\ \boxed{2}\ \boxed{*}\ \boxed{3}$$

Parentheses (and) are *word-characters* thus they are special characters considered as words in themselves and they need no blanks around them:

$$\boxed{\texttt{PRINT}}\boxed{(}\boxed{1}\ \boxed{+}\ \boxed{2}\boxed{)}\boxed{*}\ \boxed{3}$$

Peano principle: avoid grammar

A source program is a text file, whom our interpreter will parse *words* to be executed/compiled: a word is a sequence of non blanks characters delimited by blanks (some characters count as a single word, more on that later).

$$\boxed{\text{PRINT}}\ \boxed{1}\ \boxed{+}\ \boxed{2}\ \boxed{*}\ \boxed{3}$$

Parentheses (and) are *word-characters* thus they are special characters considered as words in themselves and they need no blanks around them:

$$\boxed{\texttt{PRINT}}\boxed{(}\boxed{1}\ \boxed{+}\ \boxed{2}\boxed{)}\boxed{*}\ \boxed{3}$$

Lexical analyzer

The presence of character-words implies that the delimiter of a word can be a word in itself and, once scanned as delimiter it has to be "re-scanned" as proper word.

```
_CLAST = ""

def scan_char():
    global _CLAST
    if _CLAST != "":
        c = _CLAST
        _CLAST = ""
    else:
        c = _SRC.read(1)
    return c
```

Lexical analyzer

The array _CCODES is indexed by ASCII codes and _CODES[i] is -1 if the character chr(i) is a special one, 0 if it is blank and 1 if it is part of a word.

```
_CCODES = [0]*33 + [1]*223  # ASCII only
_CCODES[ord("\n")] = -1
_CCODES[ord("(")] = -1
_CCODES[ord(")")] = -1
_CCODES[ord("\\")] = -1

def scan_word():
  global _CLAST
  while (w:=scan_char()) != "" and _CCODES[ord(w)] != -1:
    if _CCODES[ord(w)] == 1:
      while (c:=scan_char()) != "" and _CCODES[ord(c)] == 1:
        w += c
      _CLAST = c
      break
  return w
```

What do we do with a word?

Once a word w has been parsed we look for it inside a *dictionary* whose elements are quadruples (w, p, r, v) where:

- p is an unsigned byte, the *priority* of the word.
- r is the address of a machine language subroutine, the *routine* of the word.
- v is a the *word value* which is passed to the subroutine as parameter.

The higher p the sooner the word will be compiled, while the subroutine call $r(v)$ represents the operational semantics of the word.

Builtin stacks

For our purposes we will need the following stacks, during the compiling process:

- `_DICT` used to store quadruples (w, p, r, v).
- `_DSTK` used to store temporary single data (numbers/references).
- `_CSTK` used to store "compiled code" thus pairs (r, v) where r is the address of a subroutine and v a value passed it as parameter.

They'll be also available at run-time and they are, at start, empty, except for `_DICT` which contains built-in words.

Compiling and interpreting a word

Suppose we parsed w and found it in `_DICT` as (w, p, r, v):

- If $p = 0$ we call subroutine r with parameter v: a word with priority 0 is not compiled but executed at compile time.
- If $p = 255$ we immediately compile the word, thus we push the pair (r, v) on the stack `_CSTK`.
- Else, according to the value of p, we push the triple (p, r, v) on the auxiliary `_DSTK`, possibly immediately compiling words already pushed on this stack if their priorities are higher than p.

Compiling and interpreting a word

Suppose we parsed w and found it in _DICT as (w, p, r, v):

- If $p = 0$ we call subroutine r with parameter v: a word with priority 0 is not compiled but executed at compile time.
- If $p = 255$ we immediately compile the word, thus we push the pair (r, v) on the stack _CSTK.
- Else, according to the value of p, we push the triple (p, r, v) on the auxiliary _DSTK, possibly immediately compiling words already pushed on this stack if their priorities are higher than p.

Compiling and interpreting a word

Suppose we parsed w and found it in _DICT as (w, p, r, v):

- If $p = 0$ we call subroutine r with parameter v: a word with priority 0 is not compiled but executed at compile time.
- If $p = 255$ we immediately compile the word, thus we push the pair (r, v) on the stack _CSTK.
- Else, according to the value of p, we push the triple (p, r, v) on the auxiliary _DSTK, possibly immediately compiling words already pushed on this stack if their priorities are higher than p.

Compiling and interpreting a word

Suppose we parsed w and found it in _DICT as (w, p, r, v):

- If $p = 0$ we call subroutine r with parameter v: a word with priority 0 is not compiled but executed at compile time.
- If $p = 255$ we immediately compile the word, thus we push the pair (r, v) on the stack _CSTK.
- Else, according to the value of p, we push the triple (p, r, v) on the auxiliary _DSTK, possibly immediately compiling words already pushed on this stack if their priorities are higher than p.

Priorities in action

If a word w has is in the dictionary _DICT with values (p_w, r_w, v_w) and $0 < p_w < 255$, then

1. If the stack _DSTK is empty or its topmost element has priority $< p_w$ then we push (p_w, r_w, v_w) on _DSTK.
2. Else we keep on popping a triple (p, r, v) from _DSTK and pushing (r, v) on _CSTK until we are brought back to case 1, thus $p < p_w$ (or there are no more elements in _DSTK). Finally, we push (p_w, r_w, v_w) on _DSTK.

Thus, we keep aside words into _DSTK to compile them (i.e. push them on _CSTK) at the right moment.

Priorities in action

If a word w has is in the dictionary _DICT with values (p_w, r_w, v_w) and $0 < p_w < 255$, then

1. If the stack _DSTK is empty or its topmost element has priority $< p_w$ then we push (p_w, r_w, v_w) on _DSTK.
2. Else we keep on popping a triple (p, r, v) from _DSTK and pushing (r, v) on _CSTK until we are brought back to case 1, thus $p < p_w$ (or there are no more elements in _DSTK). Finally, we push (p_w, r_w, v_w) on _DSTK.

Thus, we keep aside words into _DSTK to compile them (i.e. push them on _CSTK) at the right moment.

Priorities in action

If a word w has is in the dictionary _DICT with values (p_w, r_w, v_w) and $0 < p_w < 255$, then

1. If the stack _DSTK is empty or its topmost element has priority $< p_w$ then we push (p_w, r_w, v_w) on _DSTK.
2. Else we keep on popping a triple (p, r, v) from _DSTK and pushing (r, v) on _CSTK until we are brought back to case 1, thus $p < p_w$ (or there are no more elements in _DSTK). Finally, we push (p_w, r_w, v_w) on _DSTK.

Thus, we keep aside words into _DSTK to compile them (i.e. push them on _CSTK) at the right moment.

What about unknown words?

If we parsed a word that is not in the dictionary we try to interpret it as a number, and if we succeed we compile it else we print an error message but we keep on parsing.

To compile a number N means to compile it as if the following definition would be associated to it: (`"N"`, 255, `PUSH`, N). Since it has priority 255, the pair (`PUSH,` N) is immediately compiled into `_CSTK`.

When at runtime the `PUSH(`N`)` subroutine will be executed, it'll push its argument N on the `_DSTK`, where runtime words parameters and results are stored.

I know, you need an example...

Example: `PRINT 1 + 2 * 3 - 4`

Suppose the source file is $\boxed{\texttt{PRINT 1 + 2 * 3 - 4}}$ and that the dictionary contains at least the items

- `("PRINT", 10, PRINT, NIL)`
- `("+", 100, ADD, NIL)`
- `("-", 100, SUB, NIL)`
- `("*", 110, MUL, NIL)`

We'll explain in a moment what the subroutines `PRINT`, `ADD`, `SUB` and `MUL` do, although you can make an educated guess: `NIL` is the null pointer.

Remember: we assume that at start both stacks `_CSTK` and `_DSTK` are empty.

Example: $\boxed{\texttt{PRINT}}$ `1 + 2 * 3 - 4`

Before parsing `PRINT`:

```
_CSTK = []
_DSTK = []
```

We parse $w = \texttt{PRINT}$ and find it in the dictionary with value $(p_w = 10, r_w = \texttt{PRINT}, v_w = \texttt{NIL})$.

After processing `PRINT`:

```
_CSTK = []
_DSTK = [(10, PRINT, NIL)]
```

Example: `PRINT` $\boxed{\texttt{1}}$ `+ 2 * 3 - 4`

Before parsing 1:

```
_CSTK = []
_DSTK = [(10, PRINT, NIL)]
```

We parse $w = 1$ that is not in the dictionary but it is a number, so we push the pair `(PUSH,1)` on `_CSTK`.

After processing 1:

```
_CSTK = [(PUSH, 1)]
_DSTK = [(10, PRINT, NIL)]
```

Example: PRINT 1 $\boxed{+}$ 2 * 3 - 4

Before parsing +:

```
_CSTK = [(PUSH, 1)]
_DSTK = [(10, PRINT, NIL)]
```

We parse $w = +$ and find it in the dictionary with value $(p_w = 100, r_p = \texttt{ADD}, v_w = \texttt{NIL})$.

After processing +:

```
_CSTK = [(PUSH, 1)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL)]
```

Example: `PRINT 1 +` $\boxed{2}$ `* 3 - 4`

Before parsing 2:

```
_CSTK = [(PUSH, 1)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL)]
```

We parse $w = 2$ that is not in the dictionary but it is a number, so we push the pair `(PUSH,2)` on `_CSTK`.

After processing 2:

```
_CSTK = [(PUSH, 1), (PUSH, 2)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL)]
```

Example: PRINT 1 + 2 $\boxed{*}$ 3 - 4

Before parsing $*$:

```
_CSTK = [(PUSH, 1), (PUSH, 2)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL)]
```

We parse $w = *$ and find it in the dictionary with value $(p_w = 110, r_p = \texttt{MUL}, v_w = \texttt{NIL})$.

After processing $*$:

```
_CSTK = [(PUSH, 1), (PUSH, 2)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL),
(110, MUL, NIL)]
```

Example: `PRINT 1 + 2 *` $\boxed{3}$ `- 4`

Before parsing ∗:
```
_CSTK = [(PUSH, 1), (PUSH, 2)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL),
(110, MUL, NIL)]
```

We parse $w = 3$ that is not in the dictionary but it is a number, so we push the pair `(PUSH,2)` on `_CSTK`.

After processing ∗:
```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL),
(110, MUL, NIL)]
```

Example: `PRINT 1 + 2 * 3` $\boxed{-}$ `4`

Before parsing −:
```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3)]
_DSTK = [(10, PRINT, NIL), (100, ADD, NIL),
(110, MUL, NIL)]
```

We parse $w = -$ and find it in the dictionary with value $(p_w = 100, r_p = \mathtt{SUB}, v_w = \mathtt{NIL})$.

After processing $*$:
```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3),
(MUL, NIL), (ADD, NIL)]
_DSTK = [(10, PRINT, NIL), (100, SUB, NIL)]
```

Example: `PRINT 1 + 2 * 3 -` $\boxed{4}$

Before parsing 4:

```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3),
(MUL, NIL), (ADD, NIL)]
_DSTK = [(10, PRINT, NIL), (100, SUB, NIL)]
```

We parse $w = 4$ that is not in the dictionary but it is a number, so we push the pair `(PUSH,2)` on `_CSTK`.

After processing $*$:

```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3),
(MUL, NIL), (ADD, NIL), (PUSH, 4)]
_DSTK = [(10, PRINT, NIL), (100, SUB, NIL)]
```

Example: PRINT 1 + 2 * 3 - 4

Before parsing \n:

```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3),
(MUL, NIL), (ADD, NIL), (PUSH, 4)]
_DSTK = [(10, PRINT, NIL), (100, SUB, NIL)]
```

Now the text line is ended, and the newline has the effect to pop elements from _DSTK and to push them to _CSTK until _DSTK is empty.

Finally we have:

```
_CSTK = [(PUSH, 1), (PUSH, 2), (PUSH, 3),
(MUL, NIL), (ADD, NIL), (SUB, NIL), (PRINT,
NIL)]
_DSTK = []
```

Stacks handling

Here's the Python code: stacks will be implemented as lists with elements pushed on their ends, which is trivial in Python:

```python
_CSTK = []
_DSTK = []

def push(stk, elem):
   stk.append(elem)

def pop(stk):
   exit_on(len(stk) == 0, "Missing value (stack underflow)")
   return stk.pop()
```

Compiler

```
def compile_words(n):
   while len(_DSTK) >= 3 and _DSTK[-1] >= n:
      p = pop(_DSTK)
      r = pop(_DSTK)
      v = pop(_DSTK)
      push(_CSTK, r)
      push(_CSTK, v)

def compile(p, r, v):
   if p == 0:
      r(v)
   elif p == 255:
      push(_CSTK, r)
      push(_CSTK, v)
   else:
      compile_words(p)
      push(_DSTK, v)
      push(_DSTK, r)
      push(_DSTK, p)
```

Compiler

```python
def find_word(w):
    for i in range(len(_DICT) - 4, -1, -4):
        if _DICT[i] == w:
            return i
    return -1

def compile_file():
    while (w := scan_word()) != "":
        if (i := find_word(w)) >= 0:
            compile(_DICT[i+1], _DICT[i+2], _DICT[i+3])
        else:
            try:     # probe a number (dirty)
                compile(255, PUSH, float(w))
            except ValueError:
                error_on(True, f"Unknown word {w}")
    compile_words(0)
```

Executing compiled code

After a source file has been compiled in this way, _CSTK contains a sequence of machine code calls (r, v) which can be easily executed, from the bottomost to the topmost one. We use a global variable _IP to keep track of the next instruction to execute.

```
_IP = 0

def execute():
    global _IP
    _IP = 0
    while _IP < len(_CSTK):
        _IP += 2
        _CSTK[_IP-2](_CSTK[_IP-1])
```

Example

Let us execute the previous compiled code:

```
(PUSH, 1)
(PUSH, 2)
(PUSH, 3)
(MUL, NIL)
(ADD, NIL)
(SUB, NIL)
(PRINT, NIL).
```

We need the subroutines implementations, next slide:

Executing compiled code

Runtime subroutine use the _DSTK to store parameters and result, as any traditional stack-machine: their implementations are usually straightforward (POP is a useful auxiliary routine):

```
def POP(): return pop(_DSTK)

def PUSH(v): push(_DSTK, v)

def ADD(v): PUSH(POP() + POP())

def SUB(v): PUSH(-POP() + POP())

def MUL(v): PUSH(POP() * POP())

def PRINT(v): print(POP())
```

Therefore, executing the code of the previous slide goes as in the following one:

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	PUSH(1)	[1]
4	PUSH(2)	[1 2]
6	PUSH(3)	[1 2 3]
8	MUL(NIL)	[1 6]
10	ADD(NIL)	[7]
12	PUSH(4)	[7 4]
14	SUB(NIL)	[3]
16	PRINT(NIL)	[] (3 is printed)

Executing a _CSTK

_IP	Instruction	_DSTK
2	`PUSH(1)`	`[1]`
4	`PUSH(2)`	`[1 2]`
6	`PUSH(3)`	`[1 2 3]`
8	`MUL(NIL)`	`[1 6]`
10	`ADD(NIL)`	`[7]`
12	`PUSH(4)`	`[7 4]`
14	`SUB(NIL)`	`[3]`
16	`PRINT(NIL)`	`[]` (3 is printed)

Believe it or not, we have a compiler!

WTF! Compiler, interpreter, VM

We have implemented the skeleton of a compiler/interpreter for a language whose runtime is a classical stack virtual machine, classical since Chuck Moore described it more than 20 years before Java, Python and C# and implemented it in the late 60s. (credits should due also to Dijkstra and Wirth).

This language performs *Word Translation, as in Forth*, so we call it

WTF

WTF! Compiler, interpreter, VM

We have implemented the skeleton of a compiler/interpreter for a language whose runtime is a classical stack virtual machine, classical since Chuck Moore described it more than 20 years before Java, Python and C# and implemented it in the late 60s. (credits should due also to Dijkstra and Wirth).

This language performs *Word Translation, as in Forth*, so we call it

WTF

WTF! Compiler, interpreter, VM

We have implemented the skeleton of a compiler/interpreter for a language whose runtime is a classical stack virtual machine, classical since Chuck Moore described it more than 20 years before Java, Python and C# and implemented it in the late 60s. (credits should due also to Dijkstra and Wirth).

This language performs *Word Translation, as in Forth*, so we call it

WTF

Simple example: expression evaluator

For example let us add to the dictionary some words, whose implementation are provided as simple routines to execute at runtime. The result is a simple expression language, usually the first example in any compiler book (implemented therein as a top-down parser, while our approach is a bottom-up one).

Priority	Words
0	`( )`
10	`PRINT`
60	`OR`
70	`AND`
80	`NOT`
90	`= < > <> >= <=`
100	`+ -`
110	`* /`
120	`NEG`
130	`**`
250	`ABS`
255	any number

How parentheses works

Notice that parentheses have priority 0: they are executed at compile time and do the following:

- (pushes on `_DSTK` a fake word `(0, ")", NIL)` which has priority 0 (no compiled word has it!).
-) pops items from `_DSTK` and compiles them on `_CSTK` until the fake element `(0, ")", NIL)` is found (and removed).

Since `_DSTK` is a stack, parentheses nesting is guaranteed to work the expected way!

Expression evaluator

```
def ABS(v): PUSH(abs(POP()))
def ADD(v): PUSH(POP() + POP())
def DIV(v): PUSH((1.0 / POP()) * POP())
def MUL(v): PUSH(POP() * POP())
def NEG(v): PUSH(-POP())
def POW(v):
    e = POP()
    PUSH(POP() ** e)
def SUB(v): PUSH(-POP() + POP())

# Notice: Boolean values are converted to numebers (0/1)
def EQ(v): PUSH(float(POP() == POP()))
def GEQ(v): PUSH(float(POP() <= POP()))
def GT(v): PUSH(float(POP() < POP()))
def LEQ(v): PUSH(float(POP() >= POP()))
def LT(v): PUSH(float(POP() > POP()))
def NEQ(v): PUSH(float(POP() != POP()))

def AND(v): PUSH(float(POP() != 0 and POP() != 0))
def NOT(v): PUSH(float(POP() == 0))
def OR(v): PUSH(float(POP() != 0 or POP() != 0))
```

Expression evaluator

```
def open_par(r):
    PUSH(None)
    PUSH(r)
    PUSH(0)

def close_par(m):
    while len(_DSTK) >= 3:
        p = pop(_DSTK)
        r = pop(_DSTK)
        v = pop(_DSTK)
        if m == r:
            return
        push(_CSTK, r)
        push(_CSTK, v)
    error_on(True, f"Unmatched parenthesis '{m}'")
```

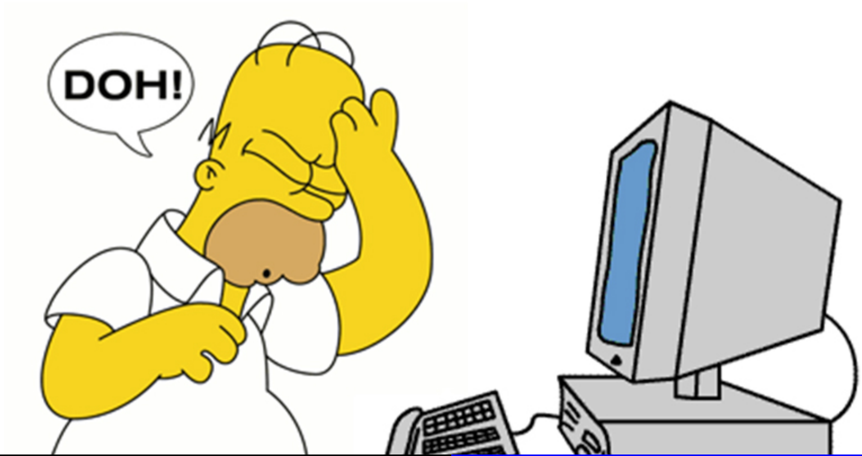
Expression evaluator

```
def COMMENT(v):
    global _NLINE
    # Skip until (and included) the next '\n'
    while (c := scan_char()) != "" and c != "\n":
        pass
    if c == "\n":
        _NLINE += 1

def NEWLINE(v):
    global _NLINE
    compile_words(0)
    _NLINE += 1

def PRINT(v):
    print(POP())
```

Expression evaluator

```
_DICT = [
    "(", 0, open_par, ")",
    ")", 0, close_par, ")",
    "*", 110, MUL, None,
    "**", 130, POW, None,
    "+", 100, ADD, None,
    "-", 100, SUB, None,
    "/", 110, DIV, None,
    "<", 90, LT, None,
    "<=", 90, LEQ, None,
    "<>", 90, NEQ, None,
    "=", 90, EQ, None,
    ">", 90, GT, None,
    ">=", 90, GEQ, None,
    "ABS", 200, ABS, None,
    "AND", 70, AND, None,
    "NEG", 120, NEG, None,
    "NOT", 80, NOT, None,
    "OR", 60, OR, None,
    "PRINT", 10, PRINT, None,
    "\\", 0, COMMENT, None,
    "\n", 0, NEWLINE, None
]
```

OK but that's a toy not even Turing-complete!



Let's add variables

Till now the only way to modify the dictionary is to do that via Python: let us remedy by introducing words to define variables.

We will store che value of variables inside a new stack `_VSTK`, whose elements are addressed by word values (thus the value of a word containing a variable will be an index to `_VSTK`.

We provide two kind of variables: simple variables (containing a number/address) and stacks.

Fetching and storing a variable's value

The following runtime routines will be used by words defining and assigning variables.

```
_VSTK = []

def VPUSH(v):
    push(_DSTK, _VSTK[v])

def VSTORE(v):
    global _VSTK
    _VSTK[v] = POP()
```

DEF name = value

At compile time:

1. Pushes 0 on _VSTK and get the address *i* of it.
2. Scan a word *w* and insert a definition (*w*, 255, VPUSH, *i*) on _DICT.
3. Scan the = raising an error if not found

At runtime:

1. Push the address *i* of the variable on the stack (VPUSH does that).
2. Push the result of the evaluation of the expression following = and store it at the location in _VSTK whose address was pushed at item 1.

Easier to express it in Python

```
_VSTK = []

def insert_word(p, r, v):
    compile_words(1)     # compile everything before definition
    w = scan_word()
    push(_DICT, w)
    push(_DICT, p)
    push(_DICT, r)
    push(_DICT, v)

def DEF(v):       # DEF word = ...
    i = len(_VSTK)       # index of the item to allocate
    push(_VSTK, 0.0)     # allocate item
    insert_word(255, VPUSH, i)
    error_on(scan_word() != "=", "'=' expected")
    compile(50, VSTORE, i)

_DICT.append("DEF")
_DICT.append(0)
_DICT.append(DEF)
_DICT.append(None)
```

LET name = value

To change a variable we need its address (mentioning it we get its value), so we use the priority 0 word `LET`.

At compile time:

1. Scan the word following it and check that its definition is stored in `_DICT` (error otherwise).
2. Scan the = raising an error if not found

At runtime:

1. Push the address of the variable on the stack (`VPUSH` does that).
2. Push the result of the evaluation of the expression following = and store it at the location in `_VSTK` whose address was pushed at item 1.

LET name = value

```
def compile_assignment(r):
    w = scan_word()
    i = find_word(w)
    if i < 0 or _DICT[i + 2] != VPUSH:
        error_on(True, f"Unknown variable {w}")
    else:
        error_on(scan_word() != "=", "'=' expected")
        compile(50, r, _DICT[i+3])
    return i

_DICT.append("LET")
_DICT.append(0)
_DICT.append(compile_assignment)
_DICT.append(VSTORE)
```

STACK name

Stacks are variables which contains neither numbers nor addresses but stacks: to keep things simple we use a different word to defined them, `STACK`.

At compile time:

1. Pushes `[]` on `_VSTK` and get the address i of it.
2. Scan a word w and insert a definition (w, 255, `VPUSH`, i) on `_DICT`.

At runtime:

1. Push the address of the variable on the stack (`VPUSH` does that).
2. Push the result of the evaluation of the expression following = and store it at the location in `_VSTK` whose address was pushed at item 1.

Easier to express it in Python

```
def STACK(v):
    i = len(_VSTK)        # index of the item to allocate
    push(_VSTK, [])       # allocate empty stack
    insert_word(255, VPUSH, i)

_DICT.append("STACK")
_DICT.append(0)
_DICT.append(STACK)
_DICT.append(None)
```

Modifying stacks

It's easy to implement words `POP`, `PUSH` and `TOS` (top of stack: retrieve the topmost element without popping it) and also the classical notation $s[i]$ to get the value of the i-th element of a stack.

We do that via two character-words [and] that use a device similar to (and).

However, to assign an element we need to invent a different notation (otherwise we could introduce states to discriminate the syntactic context of a l-value from a r-value etc. but: basic principle!) which is *index* `OF` *stack* = *value*.

Stack variables handling

```
def SPUSH(v):    # PUSH(s v)
    v = POP()
    s = POP()
    push(s, v)
def SPOP(v):    # POP(s)
    s = POP()
    PUSH(pop(s))
def STOS(v):    # TOS(s)
    s = POP()
    exit_on(len(s) == 0, "Missing data (stack underflow)")
    PUSH(s[-1])
def SLEN(v):     # LEN(s)
    s = POP()
    PUSH(len(s))

_DICT.extend(["PUSH", 10, SPUSH, None,
              "POP", 200, SPOP, None,
              "TOS", 200, STOS, None,
              "LEN", 200, SLEN. None])
```

Fetch and store items: easy, too

```
def IPUSH(v):
    i = int(POP())
    s = POP()
    exit_on(i < -len(s) or i >= len(s), "Index out of range")
    PUSH(s[i])

def ISTORE(v):
    global _VSTK
    e = POP()
    i = int(POP())
    exit_on(i < -len(_VSTK[v]) or i >= len(_VSTK[v]), "Index out
    of range")
    _VSTK[v][i] = e

def CLOSEBRA(r):
    close_par(r)
    compile(255, IPUSH, None)

_DICT.extend(["[", 0, open_par, "]",
              "]", 0, CLOSEBRA, "]",
              "OF", 0, compile_assignment, ISTORE])
```

Example

```
DEF i = 0
STACK s
PUSH(s 1)          \ Now s = [1]
s PUSH 2           \ Now s = [1 2]
PUSH s 3           \ Now s = [1 2 3]
PRINT s[1]         \ Print 2
1 OF s = 10        \ Now s = [1 10 3]
PRINT s[1]         \ Print 10
PRINT s            \ Bug: it prints the stack
```

To provide meaningful examples we need control structures such as `if`, `while` and `for` of other languages. Let's add them.

Control structures

By defining routines that mess with the _IP during execution we can easily implement control structures, as words which are executed during compilation and that compile jumps etc. writing jump addresses after the jump instruction has been compiled.

We need just two runtime jump instructions:

```
def JP(v):
    global _IP
    _IP = v

def JPZ(v):
    global _IP
    if POP() == 0:
        _IP = v
```

WTF conditionals

```
IF e
THEN
   s
ELIF e_1
THEN
   s_1
...
ELIF e_n
THEN
   s_n
ElSE
   t
FI
```

i_0	e
i_1	JPZ($i_2 + 2$)
$i_1 + 2$	s
i_2	JP(i_8)
$i_2 + 2$	e_1
i_3	JPZ($i_4 + 2$)
$i_3 + 2$	s_1
i_4	JP(i_8)
$i_4 + 2$	...
i_5	e_n
i_6	JPZ($i_7 + 2$)
$i_6 + 2$	s_n
i_7	JP(i_8)
$i_7 + 2$	t
i_8	...

Conditional implementation

The _PSTK stack contains the indexes in _CSTK where to store the jump addresses, that ELSE and FI will write: we need another stack not to mess _DSTK which is used to handle word priorities.

```
_PSTK = []  # Stack used by IF, etc. to share data

def IF(v):
    compile_words(1)     # compile everything before IF
    push(_PSTK, FI)      # FI expects this
    push(_PSTK, IF)      # THEN expects this

def THEN(v):
    error_on(pop(_PSTK) != IF, "'THEN' without 'IF'")
    # Compile expressions to _CSTK and next compile JP
    compile_words(1)
    push(_CSTK, JPZ)
    push(_CSTK, 1e20)    # changed later
    # mark where the jumping "address" will be written
    push(_PSTK, len(_CSTK) - 1)
    push(_PSTK, THEN)    # ELSE and FI expect this
```

Conditional implementation

```
def ELIF(v):
    ELSE(v)
    pop(_PSTK)
    push(_PSTK, IF)      # THEN expects this

def ELSE(v):
    error_on(pop(_PSTK) != THEN, "'ELSE' without 'THEN'")
    # Compile expressions to _CSTK and next compile JP
    compile_words(1)
    push(_CSTK, JP)
    push(_CSTK, 1e20)    # changed later
    i = pop(_PSTK)       # index where to write a jump address
    # mark where the jumping "address" will be written
    j = len(_CSTK) - 1
    push(_PSTK, j)
    _CSTK[i] = j + 1     # The JPZ compiled by THEN jumps here
    push(_PSTK, ELSE)    # FI expects this
```

Conditional implementation

```
def FI(v):
    m = pop(_PSTK)
    error_on(m != THEN and m != ELSE, "'FI' without 'THEN'/'ELSE'
    ")
    # A list of addresses where to write a pointer to the next
    # compiled instruction are written above FI in _PSTACK: they
    # are n + 1, being n the number of ELIFs
    compile_words(1)
    while (i := pop(_PSTK)) != FI:
        _CSTK[i] = len(_CSTK)

_DICT.extend(["IF", 0, IF, None,
              "THEN", 0, THEN, None,
              "ELIF", 0, ELIF, None,
              "ELSE", 0, ELSE, None,
              "FI", 0, FI, None])
```

WTF loops

```
WHILE e
DO s
OD
...

FOR w
   = e₁
   TO e₂
DO
   s
NEXT
...
```

i_0	e
i_1	`JPZ`(i_2+2)
i_1+2	s
i_2	`JP`(i_0)
i_2+2	...

i_0	e_1
i_0+2	`VSTORE`(v)
i_0+4	`VPUSH`(v)
i_0+6	e_2
i_0+8	`LT(NIL)`
i_0+10	`JPZ`(i_1+4)
i_0+12	s
i_1	`VINCR`(v)
i_1+2	`JP`(i_0+4)
i_1+4	...

Loops implementation

```
def WHILE(v):
    compile_words(1)     # compile everything before WHILE
    # mark where to jump to repeat the loop
    push(_PSTK, len(_CSTK))
    push(_PSTK, WHILE)  # DO expects this

def DO(v):
    m = pop(_PSTK)
    error_on(m != WHILE and m != FOR, "'DO' without 'WHILE' or '
    FOR'")
    # Compile expressions to _CSTK and next compile JP
    compile_words(1)
    push(_CSTK, JPZ)
    push(_CSTK, 1e20)    # changed later
    # mark where the jumping "address" will be written
    push(_PSTK, len(_CSTK) - 1)
    push(_PSTK, DO)    # OD expects this
```

Loops implementation

```
def OD(v):
    error_on(pop(_PSTK) != DO, "'OD' without 'DO'")
    # now _PSTK = [..., a, b] where a is the address of the while
    # condition and b is the address of the argument of the JPZ
    # compiled by OD: in the latter we need to write the address
    # of the first item following the loop, the former will be #
    the argument of the JP compiled by OD to repeat the loop.
    b = pop(_PSTK)
    a = pop(_PSTK)
    compile_words(5)
    push(_CSTK, JP)
    push(_CSTK, a)
    _CSTK[b] = len(_CSTK)

_DICT.extend(["WHILE", 0, WHILE, None,
              "DO", 0, DO, None,
              "OD", 0, OD, None])
```

Loops implementation

```
def FOR(v):        # FOR w = e1 TO e2 DO ... NEXT
    i = compile_assignment(VSTORE)
    push(_PSTK, _DICT[i+3]) # index of the control variable,
                            # needed later
    push(_PSTK, FOR)        # TO expects this

def TO(v):         # TO expr DO
    compile_words(1)
    j = len(_CSTK)         # location of the "TO condition": NEXT
                           # will jump here to repeat the loop
    error_on(pop(_PSTK) != FOR, "'TO' without 'FOR'")
    # compile the condition "loopvar < expr"
    i = pop(_PSTK)         # loop variable index in _VSTK
    compile(255, VPUSH, i)
    compile(50, LT, None)
    push(_PSTK, j)
    push(_PSTK, i)
    push(_PSTK, FOR)

# Compiled by NEXT (see next slide)
def VINCR(v):
    global _VSTK
    _VSTK[v] += 1
```

Loops implementation

```
def NEXT(v):
    global _CSTK
    # expect _PSTK = [ ... j i b FOR ] where j is the address
    # where the NEXT will jump to iterate the loop, i is the
    # index in _VSTK of the loop control variable, b is the
    # address of the argument of the JPZ compiled by DO where
    # the address of the first instruction following the loop
    # needs to be stored.
    error_on(pop(_PSTK) != DO, "'NEXT' without 'DO'")
    b = pop(_PSTK)
    i = pop(_PSTK)
    j = pop(_PSTK)
    # compile the increment of the loop variable
    compile(255, VINCR, i)
    # compile a jump to the condition compiled by TO
    compile(255, JP, j)
    # compile the address of the next instruction at b
    _CSTK[b] = len(_CSTK)

_DICT.extend(["FOR", 0, FOR, None,
              "TO", 0, TO, None,
              "NEXT", 0, NEXT, None])
```

Commands, procedures, functions

Our final step toward a minimal but decent programming language will be the introduction of user defined subroutines. The basic principle forbids to introduce a general construction to define any word with any priority: rather we'll allow for

- Commands, thus user defined words with priority 0.
- Procedures, thus words with priority 10 (such as `PRINT`).
- Functions, thus words with priority 250.

It'll turn out that our stack discipline for variables implies the concept of local variable with changes at all!!!

Defining a new chunk of code

We'll consider PROCedures, the case of ComManDs and FUNCtions will be similar. The syntax is

```
PROC w
      ...
END
```

`PROC` parses word w and defines a new definition $(w, 10, \mathtt{CALL}, a)$ where a is the address of a stack where the ... words are compiled. The `END` command compiles `RET` (which pairs `CALL`) and restores the compilation to `_CSTK`.

Moreover, definitions occurred inside the procedure are deleted by `END`, allowing for local data.

Example

```
PROC sort
    \ Insert sort
    DEF L = \ Local parameter, empty definition
    DEF tmp = 0
    DEF i = 0
    DEF j = 0

    FOR i = 1 TO LEN(L) DO
        LET j = i
        WHILE (IF j > 0 THEN L[j - 1] > L[j] ELSE 0 FI) DO
            LET tmp = L[j]
            j OF L = L[j - 1]
            j - 1 OF L = tmp
            LET j = j - 1
        OD
    NEXT
END
```

Call this procedure with `sort` *e*.

Procedure implementation

```
def CALL(v):
    global _IP, _CSTK
    push(_VSTK, _CSTK)  # saves on _VSTK the current _CSTK
    push(_VSTK, _IP)    # and the current _IP. RET will restore.
    _CSTK = v
    _IP = 0

def BEGIN(p):
    global _CSTK, _DICT
    # Inserts a new definition in _DICT and leaves the _PSTK as
    # [ ... c d BEGIN] where d is the current limit of _DICT and
    # c is a reference to _CSTK. d is used to "undefine" all
    # definition inside the block, while c is used to restore
    # the stack where to compile the code surrounding the block.
    # The value of the new definition is the address of the
    # block code which will be compiled until the next END word.
    # So we save _CSTK, define a new empty _CSTK pointed by the
    # new word and save also a "sentinel" BEGIN expected by END
    push(_PSTK, _CSTK)
    _CSTK = []          # now code will be compiled here
    insert_word(p, CALL, _CSTK)
    push(_PSTK, len(_DICT))
    push(_PSTK, BEGIN)  # END expects this
```

Procedure implementation

```
def RET(v):
    global _IP, _CSTK
    _IP = pop(_VSTK)
    _CSTK = pop(_VSTK)

def END(v):
    global _CSTK, _DICT
    compile_words(0)     # compile anything before END
    error_on(pop(_PSTK) != BEGIN, "'END' without 'BEGIN'")
    compile(255, RET, 0)
    # deletes all definitions local to the ending one.
    d = pop(_PSTK)  # len(_DICT) when BEGIN was executed
    while len(_DICT) > d:
        # drop the last four entries on top of _DICT
        _DICT.pop()
        _DICT.pop()
        _DICT.pop()
        _DICT.pop()
    _CSTK = pop(_PSTK)

_DICT.extend(["CMD", 0, BEGIN, 0,
              "PROC", 0, BEGIN, 10,
              "FUNC", 0, BEGIN, 250])
```

Example of function

```
FUNC fact
    DEF x =
    IF x <= 1 THEN 1
    ELSE x * fact(x - 1)
    FI
END

PRINT fact(10)
```

WTF more can we do?

- Files, source inclusions and other I/O stuff.
- Writing WTF in WTF itself: most words can be defined in terms of a few primitive ones, and the interpreter itself can be written in WTF and bootstrapped accordingly.
- Making compiled codes first class objects: for example code inside { and } could be compiled somewhere and the address returned on the stack to be used as value for a variable etc.
- Introducing structures, objects, coroutines.

But, whatever you imagine: Keep It Simple, Stupid!!!

Thanks for your patience and attention!

https://www.linkedin.com/in/paolocaressa/

https://github.com/pcaressa