

APT Session 5: Interpreters



Laurence
Tratt



Software Development Team
2016-11-11

What to expect from this session

- 1 How do programming languages run programs?
- 2 Building your own interpreter.

Prerequisites

- 1 Have the programming language of your choice (e.g. Java, Python) installed and running on your computer.

How do programming languages run programs?

Generally either by:

- 1 Compiling down into machine code at compile-time (e.g. C).

How do programming languages run programs?

Generally either by:

- 1 Compiling down into machine code at compile-time (e.g. C).
- 2 Compiling to machine code at run-time (e.g. Java).

How do programming languages run programs?

Generally either by:

- 1 Compiling down into machine code at compile-time (e.g. C).
- 2 Compiling to machine code at run-time (e.g. Java).
- 3 Having another program *interpret* your program at run-time (e.g. Python).

Interpreters

- An interpreter for language x loads in x source code and runs it.

Interpreters

- An interpreter for language x loads in x source code and runs it.
- The main steps of an interpreter are, in essence:
 - 1 Set pc (Program Counter) to 0.
 - 2 Load instruction at position pc .
 - 3 Perform the instruction loaded and adjust the pc (generally adding 1 to it).
 - 4 Jump to step #2.

Interpreters

- An interpreter for language x loads in x source code and runs it.
- The main steps of an interpreter are, in essence:
 - 1 Set pc (Program Counter) to 0.
 - 2 Load instruction at position pc .
 - 3 Perform the instruction loaded and adjust the pc (generally adding 1 to it).
 - 4 Jump to step #2.
- Although interpreters aren't particularly fast (on their own), they're fast enough that they're used heavily in the real-world.

Our language

We're going to build an interpreter for a simple stack-based language. Here's an example program:

```
INT 2    Push 2 onto the stack
INT 3    Push 3 onto the stack
ADD      Pop the last 2 elements from the stack, add them,
         and push the result onto the stack
PRINT    Peek the last element from the stack and print it
```

Terminology:

Stack A FILO (First In Last Out) list.

Push Add an element to the top of the stack.

Pop Remove the top-most element from the stack for inspection.

Peek Inspect the top-most element of the stack & don't remove it.

Parsing

```
INT 2  
INT 3  
ADD  
PRINT
```

Parsing

```
INT 2  
INT 3  
ADD  
PRINT
```

Exercises:

- 1 Put the above program into a file `p1.myl`.
- 2 Write a program which reads the file in and splits each instruction into a list of strings. You may assume that every instruction has a name and 0 or 1 arguments. The list in memory should look roughly like:

```
[["INT", "2"], ["INT", "3"], ["ADD", ""], ["PRINT", ""]]
```

A basic interpreter

We now have a list in memory along the lines of:

```
[["INT", "2"], ["INT", "3"], ["ADD", ""], ["PRINT", ""]]
```

Remember the main steps of an interpreter are, in essence:

- 1 Set *pc* (Program Counter) to 0.
- 2 Load instruction at position *pc*.
- 3 Perform the instruction loaded and adjust the *pc* (generally adding 1 to it).
- 4 Jump to step #2.

A basic interpreter

We now have a list in memory along the lines of:

```
[["INT", "2"], ["INT", "3"], ["ADD", ""], ["PRINT", ""]]
```

Remember the main steps of an interpreter are, in essence:

- 1 Set *pc* (Program Counter) to 0.
- 2 Load instruction at position *pc*.
- 3 Perform the instruction loaded and adjust the *pc* (generally adding 1 to it).
- 4 Jump to step #2.

Exercises:

- 1 Write the main loop of the interpreter, implementing only `INT`, `ADD`, and `PRINT` instructions. I suggest that all elements on the stack are stored as integers.
- 2 Run `p1.myl`

Control flow

We can currently only execute a simple linear program. We need a way of *jumping* to program locations so that we have loops e.g.:

```
INT 100  
PRINT  
INT 1  
SUB  
JGE 1
```

Control flow

We can currently only execute a simple linear program. We need a way of *jumping* to program locations so that we have loops e.g.:

```
INT 100
PRINT
INT 1
SUB
JGE 1
```

Exercises:

- 1 Implement the `SUB` instruction: it pops (in order) elements e_1 and e_2 , then pushes the result of $e_2 - e_1$ onto the stack. [NB: We didn't need to be this careful for `ADD` because addition is *commutative*.]
- 2 Implement the `JGE x` instruction. It peeks at the top-most element of the stack: if it is ≥ 0 it jumps to the instruction at position x ; otherwise it adds 1 to the PC.
- 3 Store the program above as `p2.myl` and run it.

Procedures

Jumps can build for/while loops (etc.) but not function/procedures.

0: INT 100	5: INT 1
1: CALL 4	6: SUB
2: JGE 1	7: PRINT
3: EXIT	8: SWAP
4: SWAP	9: RET

Procedures

Jumps can build for/while loops (etc.) but not function/procedures.

0: INT 100	5: INT 1
1: CALL 4	6: SUB
2: JGE 1	7: PRINT
3: EXIT	8: SWAP
4: SWAP	9: RET

Exercises:

- 1 Implement the `SWAP` instruction which swaps the two top-most elements on the stack around.
- 2 Implement the `CALL x` instruction: it pushes the `pc + 1` onto the stack and jumps to position `x`.
- 3 Implement the `RET` instruction: it pops the top-most value from the stack and jumps to that value.
- 4 Implement the `EXIT` instruction: it exits the program.
- 5 Store the program above as `p3.my1` and run it.

Labels

Jumping to numeric offsets is fragile. Labels make programs more robust:

```
        INT 100
L1: PRINT
        INT 1
        SUB
        JGE L1
```

Labels

Jumping to numeric offsets is fragile. Labels make programs more robust:

```
    INT 100
L1: PRINT
    INT 1
    SUB
    JGE L1
```

Exercises:

- 1 Allow users to define labels before an instruction and to jump to it later. Labels are text *before* a colon ':'. A line can contain both a label (before a colon) and an instruction (after a colon). [NB: Labels can be defined *after* a jump which references them.]
- 2 Store the program above as `p4.my1` and run it.

Fibonacci

The Fibonacci relation is defined thus:

$$F(n) = F(n - 1) + F(n - 2)$$

$$F(1) = 1$$

$$F(0) = 0$$

Fibonacci

The Fibonacci relation is defined thus:

$$F(n) = F(n - 1) + F(n - 2)$$

$$F(1) = 1$$

$$F(0) = 0$$

Exercises:

- 1 Write the Fibonacci program in your language. You will probably need to add `DUP` (peek at the top-most element on the stack and push a copy of it), `JEQ x` (peek at the top-most element of the stack and if it is 0 jump to pc x), and `POP` (discard the top-most element of the stack).
- 2 Store the program above as `fib.myl` and run it.

Post-session exercises

Try these (no particular order):

- Convert your interpreter to use integer constants instead of strings to represent instructions (tends to give a small speed-up).
- Rewrite your interpreter in [RPython](#) and have a working JIT!