

Exercises: arithmetic expressions and stack machines

Guy McCusker¹

¹University of Bath

A compiler and an interpreter

In this exercise we will develop

- ▶ a *compiler* which turns arithmetic expressions into sequences of instructions for a simple *stack machine*.
- ▶ an *interpreter* (or *virtual machine*) which simulates the execution of those instructions.

Arithmetic expressions

We will consider a very simple language of arithmetic expressions. It contains

- ▶ numerals: 1, 31, 9086
- ▶ binary operations on arithmetic expressions, written in Lisp syntax: (op exp1 ep2).

We'll just consider the three operations of addition, multiplication and subtraction. We won't worry about representing numbers in any kind of clever way, either.

A stack machine

Our compiler will turn an expression of this kind into a sequence of instructions for a stack machine.

The machine contains a single stack, which holds numbers.

The instructions are:

- ▶ `(push n)`, where `n` is a number. This pushes that number onto the stack.
- ▶ `add`. This pops the top two numbers off the stack, adds them together, and pushes the result onto the stack.
- ▶ `sub`. Like `add`, but it subtracts one number from the other.
- ▶ `mul`. Like `add`, but it multiplies the two numbers together.

An interpreter

We need to build an interpreter for these instructions.

Write a function `interp` which takes two arguments

- ▶ a list of instructions
- ▶ a list of numbers (the starting stack)

and returns a list of numbers which represents the state of the stack after all the instructions have been executed.

For instance, `(interp '((push 3) (push 4) add) '(7))` should return the list `(7)`.

You may assume that the program is well formed.

The `interp` function

Here's a start:

```
(defun interp (instr stack)
  (if (null instr)
      ;; no more instructions, return something
      ...
      ;; there are some instructions.
      ;; Better see what the first one is, and
      ;; do something
      ...
  )
)
```

The compiler

Our next task is to *compile* arithmetic expressions into sequences of instructions.

- ▶ for an expression that is just a number, push that number onto the stack
- ▶ for an operation, we generate the sequence of instructions for the first argument, then the sequence for the second argument, then another instruction to perform the operation.

So an expression like `(+ 3 (* 4 5))` gets compiled to

```
((push 3) (push 4) (push 5) mul add)
```

Some useful bits of Lisp

You might like to use the function `concatenate` which takes several lists and joins them together.

The `let` construct helps organise programs:

```
(let ((var1 exp1)
      (var2 exp2)
      ...
      (varn expn))
      (expression))
```

evaluates `exp1, ..., expn` and puts the values in variables `var1, ..., varn`. It then evaluates `expression`, which can use those variables

We'll cover this in more detail in later lectures.

Evaluation!

Finally, we can evaluate an expression by compiling it, passing the result to the interpreter, and seeing what the final answer is.

If our interpreter is `interp` and our compiler is `compilexp`, our evaluator is this:

```
(defun evaluate (expr)
  (car (interp (compilexp expr) ())))
)
```

Other things to try

- ▶ Change the language of expressions so that we're handling *infix* operators, i.e. we have expressions like

$(3 + (5 * 9))$

instead of

$(+ 3 (* 5 9))$

This just requires a change in the compiler.

- ▶ What about *postfix* operations, i.e. things like

$(3 (5 9 *) +)$

Take a close look at the output of the compiler in this case. You may notice it has a great similarity to the input. Can you write the compiler in a very simple way? Search the internet for "Reverse Polish" for more on this...

