Project 1: A First Stack-Oriented Language

CAS CS 320: Principles of Programming Languages

Due April 15, 2024 by 11:59PM

Stack-oriented programming languages maintain a stack for intermediate values of a computation. This paradigm is less common among modern languages, but there are still some stack-oriented languages in use.[1]

The power of stack-oriented languages is their simplicity. This makes them well-suited for a first programming language implementation, but also for more serious applications like programming embedded systems. Part of the reason there aren't many existing stack-oriented languages is that programmers will typically write their own when they need one.

In this project, you will be building an interpreter for a small stack-oriented language with subroutines and dynamically scoped variables. You will be given an EBNF grammar for the syntax of the language as well as the collection of inference rules for the reduction relation which defines the operational semantics of the language. It will then be your task to construct the following functions (these are the only functions we will test):

- ▷ `parse_prog : string -> program option`
- ▷ `update_env : env -> ident -> value -> env`
- ▷ `fetch_env : env -> ident -> value option`
- ▷ `eval_prog : stack -> env -> program -> trace`
- ▷ `interp : string -> trace option`

Once you've implemented the language, you will also have to write a few programs to interpret (it's no fun to implement a programming language and not use it). **Please read the following instructions completely and carefully.**

Part 1: Parsing

A program in our language is given by the following grammar. The start symbol of this grammar is <prog> as in the first production rule. Your implementation of `parse` should returns `None` in the case that the input string is not recognized by this grammar. **Your implementation should be whitespace agnostic** except that there should be no whitespace between digits of a number or letters of an identifier. See the notes below for further detail.

[1] See, for example, Gforth (https://gforth.org).

```
<prog>   ::= {<com>}
<com>    ::= drop | swap | dup | .
           | + | - | * | / | < | =
           | |> <ident> | # <ident>
           | ? <prog> ; | def <ident> <prog> ;
           | <ident> | <num>
<num>    ::= <digit>{<digit>}
<ident>  ::= <letter>{<letter>}
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= A | B | C | D | E | F | G | H | I | J
           | K | L | M | N | O | P | Q | R | S | T
           | U | V | W | X | Y | Z
```

In English, each rule of the above grammar is as follows.

- ▷ An *identifier* is a nonempty contiguous sequence of capital English letters. There should be no whitespace between the characters of an identifier and there must be whitespace between adjacent identifiers in a program.
- ▷ A *number* is a nonempty contiguous sequence of Arabic numerals. There should be no whitespace between the digits of a number and there must be whitespace between adjacent numbers in a program.
- ▷ A *program* is sequence of zero or more commands. Commands are not required to be separated by whitespace, nor is whitespace required within commands (e.g., `defX;` is a valid program). As previously mentioned, the exception to this is in the case of adjacent numbers or adjacent identifiers.
- ▷ A *command* is either simple or compound. A single identifier is a simple command, a single number is a simple command, and there are 13 isolated keywords which are simple commands (see above).

  There are 4 compound commands. The keyword `|>` or `#` followed by an identifier is a compound command. Note that there is no requirement that there is whitespace between the keyword and the identifier. The keyword `?` followed by a sequence of commands, followed by the keyword `;` is a compound command. The keyword `def` followed by an identifier, followed by a sequence of commands, followed by the keyword `;` is a compound command.

Part 2: Evaluation

We specify the operational semantics of our language by describing (1) what we take to be a configuration and (2) when we can say that one configuration evaluates to another configuration in a single step.

Configurations

A *configuration* is a 4-tuple $\langle\ S\ ,\ E\ ,\ T\ ,\ P\ \rangle$ consisting of:

- ▷ S, a *stack* of integer values. We represent a stack in OCaml as an `int list`.
- ▷ E, an *environment* of identifier-value mappings. A value is either an integer or a program. We represent an environment in OCaml as a `(ident * value) list`, i.e., as an association list.
- ▷ T, a *trace* of strings. We represent a trace in OCaml as a `string list`.
- ▷ P, a *program*. We represent a program in OCaml as a `command list`.

Manipulating Environments

In the rules below we use meta-functions for manipulating the environment. You will have to implement in these functions in OCaml. Below they are written update and fetch, and in OCaml we write them as `update_env` and `fetch_env`. There are several ways to implement these functions, you may implement them however you would like, as long as they satisfy the following properties.

$$\begin{aligned} \texttt{fetch\_env}\ \varnothing\ \texttt{x} &= \texttt{None} \\ \texttt{fetch\_env}\ (\texttt{update\_env}\ \texttt{e}\ \texttt{x}\ \texttt{v})\ \texttt{x} &= \texttt{Some}\ \texttt{v} \\ \texttt{fetch\_env}\ (\texttt{update\_env}\ \texttt{e}\ \texttt{y}\ \texttt{v})\ \texttt{x} &= \texttt{fetch\_env}\ \texttt{e}\ \texttt{x} \qquad (\texttt{x} \neq \texttt{y}) \end{aligned}$$

In English:

- ▷ There is no binding associated with any identifier in the empty environment.
- ▷ After updating an environment with a binding for an identifier, fetching the value for that identifier in the new environment should return that binding.
- ▷ After updating an environment with a binding for an identifier, fetching the value for *a different identifier* in the new environment should return the same binding as fetching in the original environment.

We can think the environment as a dictionary-like data structure. These are exactly the same conditions we require of such a structure.

Error Handling

In the rules below which represent errors, the resulting configuration always has the same structure: it is the same as the starting configuration, but with the empty program and with an error message err added to the top of the stack. You may use any error message you'd like, but **the first five letters must be** `"panic"`.

At a minimum, you can use the error message `"panic"` for every error rule. If you want to have some fun with the error messages, you can include additional information which expresses what caused the error, e.g., you could use `"panic: stack underflow"` for the rule dropErr.

Operational Semantics

The remaining sections contain the specification of the rules for the single-step evaluation relation over configurations. We write $C \longrightarrow C'$ to indicate that the configuration C evaluates (or reduces) to C' in a single step and we write $C \longrightarrow^\star C'$ to indicate that C evaluates to C' in multiple steps.

Your function `eval_prog` should implement multi-step evaluation given a stack, an environment and a program, starting with the empty trace. In other words, it should apply the rules below as many times as possible until a configuration is reached which cannot be further reduced. We've defined the rules such that this only happens when the program is empty. The return value of `eval_prog` is just the trace in the final configuration.

Stack Manipulation. There are several command which move and duplicate elements near the top of the stack. These facilitate the writing of subroutines which work on the top elements of the stack. For example, we can write the subroutine which doubles the value on the top of the stack as

```
def DOUBLE dup + ;
```

This subroutine duplicates the top element of the stack and then adds the top two elements.

Drop

The `drop` command removes top element of the stack. It should fail in the case that the stack is empty.

$$\frac{}{\langle\ n :: S\ ,\ E\ ,\ T\ ,\ \texttt{drop}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{drop})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{drop}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{dropErr})$$

Swap

The `swap` command exchanges the top two elements of the stack. It should fail if there are fewer than two elements on the stack.

$$\frac{}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \langle\ y :: x :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{swap})$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{swapErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{swapErr}_0)$$

Duplicate

The `dup` command adds to the top of the stack the value which is on the top of the stack. It should fail if the stack is empty.

$$\frac{}{\langle\ n :: S\ ,\ E\ ,\ T\ ,\ \texttt{dup}\ P\ \rangle \longrightarrow \langle\ n :: n :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{dup})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{dup}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{dupErr})$$

Trace

The `.` command (which is a single period symbol) removes the element on the top of the stack and adds to the front of the trace the string representation of that value. It should fail if the stack is empty. We write toString for the abstract function which returns the string representation of a given integer. You should use the OCaml function `string_of_int`.

$$\frac{}{\langle\ n :: S\ ,\ E\ ,\ T\ ,\ \texttt{.}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ \mathsf{toString}(n) :: T\ ,\ P\ \rangle}\ (\mathsf{trace})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{.}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{traceErr})$$

Push

When the command is just a number, this value is added to the top of the stack. This should never fail.

Note that it is not possible to directly push a negative number to the stack, but you can push a number and then subtract it from 0, e.g., the sequence of commands `10 0 -` adds 10 to the top of the stack, then adds 0 to the top of the stack, then replaces 0 and 10 on the top of the stack with $0 - 10$.

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \mathsf{n}\ P\ \rangle \longrightarrow \langle\ n :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{push})$$

Arithmetic Operations. There are also several commands for performing arithmetic operations on values on the top of the stack. They all behave as expected.

Add

The + command replaces the top two elements of the stack with their sum. It should fail if there are fewer than two elements on the stack.

$$\frac{}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ +\ P\ \rangle \longrightarrow \langle\ (x + y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{add})$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ +\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{addErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ +\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{addErr}_0)$$

Subtract

The - command replaces the top two elements of the stack with their difference. It should fail if there are fewer than two elements on the stack. Note the order: the second-to-top value of the stack is subtracted from the value on the top of the stack.

$$\frac{}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ -\ P\ \rangle \longrightarrow \langle\ (x - y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{sub})$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ -\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{subErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ -\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{subErr}_0)$$

Multiply

The * command replaces the top two elements of the stack with their product. It should fail if there are fewer than two elements on the stack.

$$\frac{}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ *\ P\ \rangle \longrightarrow \langle\ (x * y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{mul})$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ *\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{mulErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ *\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{mulErr}_0)$$

Divide

The / command replaces the top two elements of the stack with their division. It should fail if there are fewer than two elements on the stack. It should also fail if the second argument is 0. As for subtraction, take note of the order.

$$\frac{y \neq 0}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ /\ P\ \rangle \longrightarrow \langle\ (x/y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{div})$$

$$\frac{}{\langle\ x :: 0 :: S\ ,\ E\ ,\ T\ ,\ /\ P\ \rangle \longrightarrow \langle\ x :: 0 :: S\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{divByZero})$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ /\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{divErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ /\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{divErr}_0)$$

Less Than

The < command replaces the top two elements of the stack x and y with 1 if $x < y$ and 0 otherwise. It should fail if there are fewer than two elements on the stack.

$$\frac{x < y}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ <\ P\ \rangle \longrightarrow \langle\ 1 :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{lt}_1)$$

$$\frac{x \geq y}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ <\ P\ \rangle \longrightarrow \langle\ 0 :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{lt}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ <\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{ltErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ <\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{ltErr}_0)$$

Equals

The = command replaces the top two elements of the stack x and y with 1 if $x = y$ and 0 otherwise. It should fail if there are fewer than two elements on the stack.

$$\frac{x = y}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ =\ P\ \rangle \longrightarrow \langle\ 1 :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{eq}_1)$$

$$\frac{x \neq y}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ =\ P\ \rangle \longrightarrow \langle\ 0 :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{eq}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ =\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{eqErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ =\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{eqErr}_0)$$

Variables and Subroutines. The commands for defining variables and subroutines are compound commands and, in addition to manipulating the stack, they add and fetch identifiers in the environment.

Variable Assignment

The `var` ID command updates the environment by binding the identifier ID to the value on the top of the stack. It also removes the element on the top of the stack. It should fail if the stack is empty.

$$\frac{}{\langle\ n :: S\ ,\ E\ ,\ T\ ,\ \texttt{|>}\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ S\ ,\ \mathsf{update}(E, \mathsf{ID}, n)\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{var})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{|>}\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{varErr})$$

Variable Fetch

If the command is a single identifier, then if it is bound to a number in the environment, that number is added to the top of the stack. It should fail if the identifier is not bound to a number. This occurs if the identifier is either not bound or it is bound to a program (i.e., it is the identifier of a subroutine).

$$\frac{\mathsf{fetch}(E, \mathsf{ID}) \text{ is a number}}{\langle\ S\ ,\ E\ ,\ T\ ,\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ \mathsf{fetch}(E, \mathsf{ID}) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{fetch})$$

$$\frac{\mathsf{fetch}(E, \mathsf{ID}) \text{ is not a number}}{\langle\ S\ ,\ E\ ,\ T\ ,\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{fetchErr})$$

Subroutine Definition

The `def` ID Q `;` command binds the identifier ID to the program Q in the environment. This command should never fail. Note that if the identifier is ill-formed or the body of the subroutine is ill-formed, this should be caught during parsing.

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{def}\ \mathsf{ID}\ Q\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ S\ ,\ \mathsf{update}(E, \mathsf{ID}, Q)\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{def})$$

Subroutine Call

The `#` ID command fetches the value to which ID is bound, and if it is a program, it is prepended to the existing program. It should fail if ID is not bound to a program. This occurs if ID is not bound or it is bound to a number.

$$\frac{\mathsf{fetch}(E, \mathsf{ID}) \text{ is a program}}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{\#}\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ \mathsf{fetch}(E, \mathsf{ID})\ P\ \rangle}\ (\mathsf{call})$$

$$\frac{\mathsf{fetch}(E, \mathsf{ID}) \text{ is not a program}}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{\#}\ \mathsf{ID}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{callErr})$$

This is a very simple notion of a subroutine: calling a subroutine means running a fixed sequence of commands. It is because of this fact that our variables are dynamically scoped. Evaluating the program

```
2 |> X
def F 3 |> X ;
#F X .
```

should result in the trace `["3"]` because calling `F` simply prepends the commands `3 |> X` to the program, which will update the value of `X` in the environment before it is traced. Formally, we have

$$
\begin{aligned}
&\langle\ \varnothing\ ,\ \varnothing\ ,\ \varnothing\ ,\ \texttt{2 |> X def F 3 |> X ; \#F X .}\ \rangle \\
&\longrightarrow^{\star} \langle\ \varnothing\ ,\ \mathsf{update}(\varnothing, \mathsf{X}, 2)\ ,\ \varnothing\ ,\ \texttt{def F 3 |> X ; \#F X .}\ \rangle \\
&\longrightarrow \langle\ \varnothing\ ,\ \mathsf{update}(\mathsf{update}(\varnothing, \mathsf{X}, 2), \mathsf{F}, \texttt{3 |> X})\ ,\ \varnothing\ ,\ \texttt{\#F X .}\ \rangle \\
&\longrightarrow \langle\ \varnothing\ ,\ \mathsf{update}(\mathsf{update}(\varnothing, \mathsf{X}, 2), \mathsf{F}, \texttt{3 |> X})\ ,\ \varnothing\ ,\ \texttt{3 |> X X .}\ \rangle \\
&\longrightarrow^{\star} \langle\ \varnothing\ ,\ \mathsf{update}(\mathsf{update}(\mathsf{update}(\varnothing, \mathsf{X}, 2), \mathsf{F}, \texttt{3 |> X}), \mathsf{X}, 3)\ ,\ \texttt{"3"} :: \varnothing\ ,\ \epsilon\ \rangle
\end{aligned}
$$

Control Flow. We include one compound command for control flow which will allow us to write more interesting programs. As with the numerical predicates above, we use the number 0 to represent falsity and any nonzero value to represent truth.

If-Statement

The `?` Q `;` command prepends the program Q to the current program if the value on the top of the stack is nonzero, and does nothing (except consume the command) otherwise. In either case, the value on the top of the stack is also removed. It should fail if the stack is empty.

$$\frac{}{\langle\ 0 :: S\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{if}_0)$$

$$\frac{n \neq 0}{\langle\ n :: S\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ Q\ P\ \rangle}\ (\mathsf{if}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle}\ (\mathsf{ifErr}_0)$$

Interpretation. The last step of evaluation is defining the full interpretation function. Once you've completed the `eval_prog` function, the `interp` function combines parsing and evaluation, returning `None` only when there is a parse error.

Part 3: Writing Programs

Generally speaking, I recommend trying to write a lot of programs in this language. Not only will this be interesting and fun (hopefully), but it will also help you verify that you've implemented everything correctly. To run programs, you need to uncomment the line

```
let _ = main ()
```

in the starter code. You can then write a program in a separate file `file.my` and pass it to your interpreter via input redirection on the command line:

```
utop interp_01.ml < file.my
```

An Example: Factorial

Below is an example of a subroutine which implements the factorial function. It returns -1 on negative inputs.

```
def ISNEG
  0 swap <
;

def DECR
  1 swap -
;

def FACTORIAL
  dup #ISNEG ? drop 1 0 - ;
  dup 0 = ? drop 1 ;
  dup 1 < ?
    dup
    #DECR
    #FACTORIAL
    *
  ;
;

3 #FACTORIAL .
```

Note that we do not pass arguments explicitly to a subroutine when calling it. Instead, we put the parameters on the stack and then call the subroutine. In the last line, we put 3 on the stack, call the factorial subroutine, and then trace the output. If you run this program, you should see the value `6`, or if you copy this program into a string `p` the following assertion should hold.

```
let test = interp p
let out = Some ["6"]
let _ = assert (test = out)
```

Similarly, in the first line[2] of the factorial subroutine, we duplicate the top of the stack (the parameter to the factorial subroutine) and then call the negativity-check subroutine. That is, the first line checks if the input to the factorial subroutine negative and then replaces the top of the stack with -1 in this case.

Also note that this implementation of the factorial function is recursive. Because our notion of a subroutine is so simple, we can write recursive functions essentially for free. The last conditional in the body of the factorial subroutine copies the parameter on the top of the stack, decrements it, and then calls the factorial function on it. Finally, it multiplies the result with the original input parameter.[3]

You should try to convince yourself that this makes sense. Consider drawing out the first couple steps of evaluating the program.

We could also implement this using variables, but we have to be a bit careful because of the dynamic scoping. In the following implementation, we do not handle the case of negative inputs.

```
def FACTHELP
  dup 1 < ?
    dup OUT * |> OUT
    #DECR
    #FACTHELP
  ;
;

def FACTORIAL
  1 |> OUT
  #FACTHELP
  drop OUT
```

[2]Remember that the lines are only for readability, your implementation should be whitespace agnostic.
[3]Remember that, because we are working with a stack, we need to apply operations in reverse.

```
;

3 # FACTORIAL.
```

Because subroutines are just lists of commands, we can refer to variables which are not in the environment in the body of a subroutine. We can then put the variable in the environment before calling the subroutine. The helper subroutine multiplies the variable `OUT` by the value on the top of the stack, decrements the top of the stack, and then recurses. It does this until the value on the top of the stack is 1.

Required Programs. Your last task is to write two programs in the language we've just defined.

- ▷ Fill in the subroutine `SQDIST` in `sq_dist.my` which, given that there are two elements x and y on the top of the stack (with x on top), removes them both and adds $x^2 + y^2$ on the top of the stack.
- ▷ Fill in the subroutine `FIB` in `fib.my` which, given that there is a nonnegative integer n on the top of the stack, replaces it with the nth Fibonacci number. For fun, you can write a version which maintains constant stack space. You will find that this implementation is significantly faster.

Final Remarks

- ▷ There is a lot of repetition here, this is just the nature of implementing programming languages. So even though there is a lot of code to write, it should go pretty quickly.
- ▷ Despite the repetition, it may be worthwhile to think about how to implement the interpreter without too much code reproduction.
- ▷ You don't need to understand how to program in this language to be able to build the interpreter. As long as you understand how to read a grammar and write the conversion rules above as OCaml code, you can focus on this before thinking about how to program in a stack-oriented setting.
- ▷ Test along the way. Particularly for parsing, build small parsers and test them before using them in more complex parsers.
- ▷ You are not required to use the starter code given. You are also welcome to add anything to the given starter code. **The only things which must remain the same are the names of the functions you are required to implement and any types they depend on.**