Project 2: A Language with Lexically Scoped Variables

CAS CS 320: Principles of Programming Languages

Due April 22, 2024 by 11:59PM

Introduction

In the first project, subroutines did not have local variables. If we wanted to write a program with recursion, we had to be careful about using variables that subsequent recursive calls could change. We may have been tempted to write the factorial function (on positive integers) as follows.

```
def FACTORIAL
  |> N
  N 1 = ? 1 ;
  N 1 < ? 1 N - #FACTORIAL ;
  N *
;

10 #FACTORIAL .
```

This implementation attempts to call `FACTORIAL` recursively and multiply the result by the input value, to which `N` was bound. But since variables are dynamically scoped in this language, the recursive call to `FACTORIAL` changes the value that `N` is bound to, giving us an incorrect implementation.

For this project, we will update our semantics to allow for lexically scoped variables so that our subroutines can have local variables. We will then be able to write a program like this one, which implements the same logic as above (extended to handle all integers).

```
(factorial): |> n
  (if) 0 n < ?
    Return ;
  (else if) 0 n = ?
    1 Return ;
  (else)
    -1 n + factorial #
    n *
    Return ;
  ;
; |> factorial

10 !factorial #
```

This project will be similar to the previous one. You will be given the syntax and the semantics of the language, and then it will be your task to implement a parser and an evaluator. All together you will need to construct the following functions. We have provided starter code but you are not required to use it, as long as your solution implements these functions according to the specification given in this document.

- ▷ `parse_prog : string -> program option`
- ▷ `eval_prog : program -> trace`

Part 1: Parsing

A program in our language is given by the following grammar. The start symbol is <prog> as in the first production rule. Your implementation should return `None` if the input string is not recognized by this grammar, and should be whitespace agnostic except in the case of numbers and identifiers (as in the first project).

```
<prog>  ::= {<com>}
<com>   ::= <const> | + | * | /
          | && | || | ~ | < | =
          | ? <prog> ; <prog> ;
          | While <prog> ; <prog> ;
          | <ident> | |> <ident>
          | : <prog> ; | # | Return
<const> ::= <bool> | <int>
<bool>  ::= True | False
<int>   ::= [ - ]<digit>{<digit>}
<ident> ::= <letter>{<letter> | <digit> | _}
<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
```

Notes.

- An *identifier* is a nonempty contiguous sequence of lower case English letters, underscores and numbers, starting with a letter. There should be no whitespace between characters of an identifier and there must be whitespace between adjacent identifiers.
- An *integer* is a nonempty contiguous sequence of Arabic numerals, beginning optionally with a minus sign to indicate a negative integer. There should be no whitespace between the digits of an integer or its minus sign (if applicable) and there must be whitespace between adjacent numbers in a program.
- A *program* is a sequence of commands. Commands are not required to be separated by whitespace except in the previously mentioned cases.
- **Comments.** In order to make our programs more readable, we also allow for comments in our program. A comment is any sequence of characters surrounded by parentheses. A parser for comments is given in the starter code. In particular, we augment the whitespace parser (and the keyword parser) to include comments as part of what it ignores. Note that, since the parser looks for the *closest* closing parenthesis, it is not possible to nest parentheses in a comment.
- **Debug Statements.** We also include an unofficial construct in our language which allows us to write general print statements for debugging. A debug statement is any sequence of characters between double quotations. A parser for debugging statements is given in the starter code.

Both of these last constructs are required for your submission. Again, they are given in the starter code, please just make sure not to forget them. Here is an example of a function which prints `not equal` when its inputs are not equal integers.

```
(assert): |> x |> y (function name comment is for readability)
  (if) x y = ~ ?  (control flow comments are for readability)
    "not equal" ;
  (else)
    (do nothing) ;
; |> assert          (last line actually binds the name)
```

Part 2: Evaluation

The semantics of our new language deviates from that of the first project in many important ways:

- ▷ The stack holds not just integers, but also Boolean values and closures. A name can then be bound to a function in the same way it can be bound to an integer.
- ▷ The stack is only used for parameter passing and intermediate computation. When a function is called, the stack must be cleared (except for possibly the return value) before the function returns control.
- ▷ Our environment will simulate the call stack by keeping track of local variable bindings for function calls. This requires us to take care of variable capture. See the section below for details.

Configurations

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

- ▷ S, a *stack* of values, which may be integers ($\mathbb{Z}$), Boolean values ($\mathbb{B}$), or closures ($\mathbb{C}$). We represent the stack in OCaml as a `value list`. See section below for more details about closures.
- ▷ E, an *environment* of variable bindings and call stack information. See below for details of our representation in OCaml.
- ▷ T, a *trace* of strings. We represent the trace in OCaml as a `string list`.
- ▷ P, a *program*. We represent a program in OCaml as a `command list`.

Error Handling

We are using a slightly simpler semantics for error handling. In errors rules, the resulting configurations always consist of the empty stack, empty environment, empty program, and the same trace with an additional error message. 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. We use panic to represent the configuration

$$\langle\ \varnothing\ ,\ \varnothing\ ,\ \mathsf{err} :: T\ ,\ \epsilon\ \rangle$$

below, where T is the trace from configuration on the left side of the reduction. See the function `panic` in the starter code.

Functions

Functions are represented on the stack as *closures*, which are subroutines with additional information about how and when the functions were defined. A closure is a 3-tuple $[\ i\ ,\ B\ ,\ P\ ]$ (written with square brackets) consisting of

- ▷ i, an integer-valued identifier. This identifier should be the same as the identifier of the activation record for the scope in which the function is defined. We need this value to know where to look for bindings accessible to the function.

- ▷ B, a list of captured variable bindings. In the case that a function is returned by another function, the variables in that scope are thown away. We need to copy those bindings in the closure of the returned function so they are still accessible when the function is called. We represent in OCaml as a `(ident * value) list`
- ▷ P, a program which contains the commands in the definition of the function, represented in OCaml as a `command list`.

Note that, in the starter code, we represent a closure as a record type.

The Environment

The environment in this project will simulate the call stack to implement lexical scoping. This will require a more complex notion of fetching and updating the environment. The environment is represented abstractly as a nonempty stack

- ▷ whose bottom element (written abstractly as global) and is a list of variable bindings, and
- ▷ whose remaining elements are called *activation records*, which consist of an integer-valued identifier, a list of bindings, and a program.

We write an activation record as a closure [i , B , R] because it has the same data, but the data are interpreted differently:

- ▷ i is the identifier of the called function who necessitated the activation record.
- ▷ B is the list of variable bindings local to the function call.
- ▷ R is the part of the program of the caller, which should be resumed once the called function returns control.

Each activation record also has an associated identifier. It is not represented here because **it is always the number of positions away from** global in the stack (global has identifier 0).

As an example, consider the following program.

```
-1 |> x
2 |> y
(g):
  1 |> x
;
(f):
  0 |> x
  0 |> y
  g #
  3 |> x
;
f #
2 |> y
```

which would be akin to the following Python program.

```
x = -1
y = 2
def g():
    x = 1
def f():
    x = 0
    y = 0
    g()
```

```
    x = 3
f()
y = 2
```

Calling `f` should subsequently call `g`, and just before returning from the call to `g`, we should have the following environment.

$$\begin{aligned}&[\ 0\ ,\ [(\mathsf{x},1)]\ ,\ \texttt{3 |> x}\ ]\\ &::[\ 0\ ,\ [(\mathsf{x},0);(\mathsf{y},2)]\ ,\ \texttt{2 |> y}\ ]\\ &::[(\mathsf{x},-1);(\mathsf{y};2);(\mathsf{g},[\ 0\ ,\ \varnothing\ ,\ \texttt{...}\ ]);(\mathsf{f},[\ 0\ ,\ \varnothing\ ,\ \texttt{...}\ ])]\end{aligned}$$

where the bottom element is global. Locally (i.e., within the record with identifier 2) the variables `x` is bound to 1, but when `g` returns we will drop the record and continue to bind `x` to the value 3, but in the following record (with identifier 1).

Assign. Binding a name to a value should always be done in local scope. That it, the binding should be made within the local bindings of the topmost record, or global if there are no records. Note that this means there is no way for a function to affect the value of a variable in global (this is consistent with the behavior of Python).

Fetch. When fetching a binding you should use the following procedure.

- ▷ Check the local binding in the topmost record. If the binding is there, then return it.
- ▷ If the binding is not there, then **check the record whose identifier is the same as the identifier of the called function stored in the topmost record.** Remember that the identifier of a record is its position in the stack (0 for global).
- ▷ Repeat the above process until you've reached global. If the binding isn't there, then the fetch operation failed.

In the example environment above, fetching the value of `y` in the local environment means first checking the local bindings and then checking *the bindings in the record with identifier 0*, i.e., the global scope. Even though `y` is bound in the record with identifier 1, it is not checked. This should coincide with the intuition that the binding of `y` in the `f` is not in scope when calling `g` in `f`.

In the starter code, the representation of the environment is more explicit. There is an algebraic data type for distinguishing between global and local scopes, and activation records are represented as OCaml records. Also note that activation records hold their own identifiers in the field `id`. This should making coding the assignment and fetching functions a easier but **You have to remember to set the identifier when you add a record the environment.**

Operational Semantics

The operational semantics for many of the constructions in our language are nearly identical to those in the first project. We include the rules here without any additional information. For new constructs we will include some

Push

Note that when a function is defined, its closure identifier is the identifier of the topmost record. You can use the function `local_id` given in the starter code to get this identifier. The function also has not yet captured any variables.

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{c}\ P\ \rangle \longrightarrow \langle\ c :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\textsf{push})$$

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{:}\ Q\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ [\ |E|-1\ ,\ \varnothing\ ,\ Q\ ] :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\textsf{pushFun})$$

Add

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

$$\frac{x \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{+}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_1) \qquad \frac{y \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{+}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{+}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{+}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_4)$$

Multiply

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

$$\frac{x \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{*}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_1) \qquad \frac{y \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{*}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{*}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{*}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_4)$$

Divide

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

$$\frac{x \in \mathbb{Z} \qquad y \in \mathbb{Z} \qquad y = 0}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{/}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divByZero}) \qquad \frac{x \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{/}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_1)$$

$$\frac{y \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{/}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_2) \qquad \frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{/}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_3)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{/}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_4)$$

And

We write '$\wedge$' for the logical-and operator. You can use `&&` in OCaml.

$$\frac{x \in \mathbb{B} \qquad y \in \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{\&\&}\ P\ \rangle \longrightarrow \langle\ (x \wedge y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{and})$$

$$\frac{x \notin \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{\&\&}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{andErr}_1) \qquad \frac{y \notin \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{\&\&}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{andErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{\&\&}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{andErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{\&\&}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{andErr}_4)$$

Or

We write '$\vee$' for the logical-or operator. You can use `||` in OCaml.

$$\frac{x \in \mathbb{B} \qquad y \in \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{||}\ P\ \rangle \longrightarrow \langle\ (x \vee y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{or})$$

$$\frac{x \notin \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{||}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{orErr}_1) \qquad \frac{y \notin \mathbb{B}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{||}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{orErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{||}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{orErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{||}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{orErr}_4)$$

Not

We write '$\neg$' for the logical-negation operator. You can use `not` in OCaml.

$$\frac{x \in \mathbb{B}}{\langle\ x :: S\ ,\ E\ ,\ T\ ,\ \texttt{~}\ P\ \rangle \longrightarrow \langle\ \neg x :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{not})$$

$$\frac{x \notin \mathbb{B}}{\langle\ x :: S\ ,\ E\ ,\ T\ ,\ \texttt{~}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{notErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{~}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{notErr}_2)$$

Less Than

$$\frac{x \in \mathbb{Z} \qquad y \in \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{<}\ P\ \rangle \longrightarrow \langle\ (x < y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{lt})$$

$$\frac{x \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{<}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_1) \qquad \frac{y \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{<}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{<}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{<}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_4)$$

Equals

$$\frac{x \in \mathbb{Z} \qquad y \in \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{=}\ P\ \rangle \longrightarrow \langle\ (x = y) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{eq})$$

$$\frac{x \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{=}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_1) \qquad \frac{y \notin \mathbb{Z}}{\langle\ x :: y :: S\ ,\ E\ ,\ T\ ,\ \texttt{=}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_2)$$

$$\frac{}{\langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{=}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{=}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_4)$$

If-Else

$$\frac{}{\langle\ \mathsf{True} :: S\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q_1\ \texttt{;}\ Q_2\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ Q_1\ P\ \rangle}\ (\mathsf{ifTrue})$$

$$\frac{}{\langle\ \mathsf{False} :: S\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q_1\ \texttt{;}\ Q_2\ \texttt{;}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ Q_2\ P\ \rangle}\ (\mathsf{ifFalse})$$

$$\frac{x \notin \mathbb{B}}{\langle\ x :: S\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q_1\ \texttt{;}\ Q_2\ \texttt{;}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{notErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{?}\ Q_1\ \texttt{;}\ Q_2\ \texttt{;}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{notErr}_2)$$

While

This is the only command for which we will not provide an explicit set of rules. The behavior of

$$\texttt{While}\ Q_1\ \texttt{;}\ Q_2\ \texttt{;}\ P$$

is as follows.

- ▷ The commands in Q_1 are evaluated.
- ▷ If the top of the stack after evaluating Q_1 is `False`, then the while loop is over, the top of the stack is removed and the commands in P are evaluated.
- ▷ Otherwise, if the top of the stack after evaluating Q_1 is `True`, then the top of the stack is removed and the commands in Q_2 are evaluated. Then we go back to the first bullet point and repeat.

Hint. Try to define the semantics of while-loops in a single line by replacing it with an If-Then command.

Fetch

Fetching by done simply writing down an identifier. The rules for fetching variables from the environment are nearly identical to those of the first project, but the semantics are very different because of how fetch is defined.

$$\frac{\mathsf{fetch}(E, \mathsf{x}) = v \qquad v \in \mathbb{V}}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{x}\ P\ \rangle \longrightarrow \langle\ v :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{fetch})$$

$$\frac{\mathsf{fetch}(E, \mathsf{x}) = \bot}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{x}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{fetchErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{x}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{fetchErr}_2)$$

Assign

The rules for assigning variables in the environment are nearly identical to those of the first project, but the semantics are very different because of how update is defined.

$$\frac{}{\langle\ v :: S\ ,\ E\ ,\ T\ ,\ \texttt{|>}\ \texttt{x}\ P\ \rangle \longrightarrow \langle\ S\ ,\ \mathsf{update}(E, \mathsf{x}, v)\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{assign})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{|>}\ \texttt{x}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{assignErr})$$

Call

In order to call a function, we have to put a closure on the stack. We can do this either by defining a function directly or by binding a name to a function and then fetching the value bound to that name. If there is a closure on the stack, calling the function requires the following updates to the configuration.

- ▷ Set the program being evaluated to the one given by the closure.
- ▷ Set the local environment to include the variables captured by the closure.
- ▷ Set the return program to be the remainder of the program after the call command.
- ▷ Include in the environment the closure identifer of the function being called. This tells us where to look for bindings if we can find them in local scope. Remember that we need to look where the function was *defined* not where it was called.

These last three steps are part of the activation record of the function being called. Recall that they are represented in our operational semantics as a closure, but in the starter code it is represented as a record type.

$$\frac{}{\langle\ [\ i\ ,\ B\ ,\ Q\ ] :: S\ ,\ E\ ,\ T\ ,\ \#\ P\ \rangle \longrightarrow \langle\ S\ ,\ [\ i\ ,\ B\ ,\ P\ ] :: E\ ,\ T\ ,\ Q\ \rangle}\ (\mathsf{call})$$

$$\frac{x \notin \mathbb{C}}{\langle\ x :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{callErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \#\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{callErr}_2)$$

Return

In order to finish running a function we have to return control to the caller. This requires the following updates to the configuration. Note that we allow functions to not have return statements, as long as there is no return value on the stack. We also require that return statements appear only within functions.

- ▷ Set the program being evaluted to the return program in the activation record.
- ▷ If there is there is no return statement, verify that the program being evaluated is empty and that the stack is empty.
- ▷ If there is a return statement, verify that there is exactly one value (the return value) on the stack.
- ▷ If a closure is being returned and it was defined in local scope (i.e., its closure identifier is the same the identifier of the topmost record, add the local bindings to the captured variables of the returned closure. We represent this abstractly with the function merge. **Make sure to shadow the added local bindings with the bindings which are already captured.** Also set the closure identifier to the callee identifier stored in the activation record.
- ▷ Throw away the topmost record, returning control to the caller of the function.

These steps are captured by the following rules.

$$\frac{i = |E| - 1}{\langle\ [\ i\ ,\ B\ ,\ Q\ ] :: \varnothing\ ,\ [\ j\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \langle\ [\ j\ ,\ \mathsf{merge}(B, L)\ ,\ Q\ ] :: \varnothing\ ,\ E\ ,\ T\ ,\ R\ \rangle}\ (\mathsf{ret}_1)$$

$$\frac{i \neq |E| - 1}{\langle\ [\ i\ ,\ B\ ,\ Q\ ] :: \varnothing\ ,\ [\ j\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \langle\ [\ i\ ,\ B\ ,\ Q\ ] :: \varnothing\ ,\ E\ ,\ T\ ,\ R\ \rangle}\ (\mathsf{ret}_2)$$

$$\frac{x \notin \mathbb{C}}{\langle\ x :: \varnothing\ ,\ [\ i\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \langle\ x :: \varnothing\ ,\ E\ ,\ T\ ,\ R\ \rangle}\ (\mathsf{ret}_3)$$

$$\frac{}{\langle\ \varnothing\ ,\ [\ i\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ T\ ,\ R\ \rangle}\ (\mathsf{ret}_4)$$

$$\frac{}{\langle\ \varnothing\ ,\ [\ i\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \epsilon\ \rangle \longrightarrow \langle\ \varnothing\ ,\ E\ ,\ T\ ,\ R\ \rangle}\ (\mathsf{ret}_5)$$

$$\frac{}{\langle\ x :: y :: S\ ,\ [\ i\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{retErr}_1)$$

$$\frac{}{\langle\ x :: S\ ,\ [\ i\ ,\ L\ ,\ R\ ] :: E\ ,\ T\ ,\ \epsilon\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{retErr}_2)$$

$$\frac{}{\langle\ S\ ,\ \mathsf{global}\ ,\ T\ ,\ \texttt{Return}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{retErr}_3)$$

Finally, the function `eval_prog` should evaluate the program on the empty stack, empty environment, and empty trace. See the starter code for a suggested outline.

Final Remarks

- ▷ Compared to the previous project, the difference between the code and the rules is greater. Make sure to understand how the two coincide before you start working on the code.
- ▷ You are not required to write programs in this language, but I still recommend doing it. This is going to be the way that you verify your solution works as intended.
- ▷ The starter code is just a suggestion, you are not required to use it.
- ▷ This is a difficult project, probably the most difficult of the three. We will see examples in class on April 16, so if you're confused, please be sure to attend.