EXP01:
α language interpreter

This document describes an interactive Read-Eval-Print-Loop (REPL) for the α language, described below. The REPL reads an expression, evaluates it, pushes the result onto a stack, and prints the contents of the stack.[1]

The prompt of the repl is '`repl> `' (note the space after the '>'). The repl print the contents of the stack after each full line of code it evaluates (except if an incomplete expression is entered, in which case the prompt changes to '`repl+ `'). If input is encountered that is not handled by the rules listed above, :error: is pushed onto the stack and evaluation continues with the next expression. Expressions must be separated by whitespace in the input stream (e.g. space, newline, tab).

The language handles the following expressions:

- Numbers (sequences of digits, possibly with a leading '-') are expressions. A number expression is evaluated to its base 10 interpretation. A digit sequence with a leading '-' denotes a negative number (or zero, though '-0' must be represented as '0'). A digit sequence without a leading '-' denotes a positive number (or zero). The value is pushed onto the stack.
- Boolean literals (`:true:` and `:false:`) are expressions. `:true:` evaluates to `:true:`, and `:false:` to `:false:` The value is pushed onto the stack.
- Error literal (`:error:`) is an expression. `:error:` evaluates to `:error:` The value is pushed onto the stack.
- String literals are expressions. A string consists of a sequence of printable and whitespace characters enclosed in double quotation marks, as in `"this is a string"`. The only such character which may NOT appear inside a string is the double quote character; other characters, including the newline character, can appear inside a string. A string expression evaluates to itself. The value is pushed onto the stack.
- Names (sequences of letters and digits, starting with a letter) are expressions. A name expression is evaluated by looking the name up relative to a *list* of environments. The first environment in that list is referred to as the current environment. A name has two possible interpretations, a *local-only* value and a *not-only-local* value:
    - The *local-only* value is determined by looking up the name in the current environment ONLY. If the name is not bound in the current environment then the value of the name is itself; otherwise the value of the name is the value it is bound to. Remember that this value can be ANY type of expression, including a name.
    - The *not-only-local* value is determined by looking up the name in the full list of environments, starting with the current environment and continuing on down the

[1] There is some additional partial functionality in the code, to do with closures and their application. For the purposes of this project you can ignore those parts of the code, as well any bugs or missing functionality stemming from that code. None of the bugs (or feature requests) given here are impacted by the closure or closure application code.

list of environments. If the name is not bound in ANY of the environments then the value of the name is itself; otherwise the value of the name is the FIRST value found in searching the environments, starting with the current environment. Remember that this value can be ANY type of expression, including a name.

The value of the name expression is the pair of its *local-only* value and its *not-only-local* value: <*local-only*, *not-only-local*>. We need this pair of possible interpretations in order to allow for local bindings of names which are bound non-locally:

> 'bind' will always choose the local-only value of the name. If the *local-only* value is a name the binding will succeed. If the *local-only* value is not a name then binding will result in an error.
>
> Any operation other than 'bind' will access the *not-only-local* value.

We call a <*local-only*, *not-only-local*> pair a *value pair*.

- Applications of primitive numeric functions. Only these primitive numeric functions are supported: `add`, `sub`, `mul`, `div`, `rem` and `neg`. They refer to the integer addition, subtraction, multiplication, division, remainder and negation operations.
    - The primitive operators `add`, `sub`, `mul`, `div` and `rem` are binary and therefore consume the top two values from the stack, pushing the result[2] (a single value) onto the stack. If there are not two values on the stack, or if the two values on the stack are not both numbers, the value `:error:` is pushed onto the stack. In the case that an error results[3], any values popped off the stack must be pushed back on the stack, in the same order, before `:error:` is pushed onto the stack.
    - The operator `neg` is unary and therefore consumes the top value from the stack, pushing the result (a single value) onto the stack. If there isn't a numeric value on the stack (i.e. the stack is empty or a non-number is on the stack) the value `:error:` is pushed onto the stack. In the case that an error results, any value popped off the stack must be pushed back on before `:error:` is pushed onto the stack.
    - The behavior of `rem` is given by this description:

      if we denote (x y div) by q, and (x y rem) by r, the definition given in class was that q and r are the unique integers such that both:

$$x=qy+r \text{ and } 0\leq r<|y|$$

- Applications of primitive Boolean functions. Only these primitive Boolean functions need be supported: `and`, `or` and `not`. They refer to logical conjunction, logical disjunction and logical negation. A primitive function is applied by calling 'native' code to perform the operation (i.e. code you define in your interpreter).
    - The primitive operators `and` and `or` are binary and therefore consume the top two values from the stack, pushing the result (a single value) onto the stack. If there

[2] `x y sub` computes x-y; `x y div` computes x/y; `x y rem` computes the remainder of x/y.

[3] Including a divide-by-zero error for `div`.

are not two values on the stack, or if the two values on the stack are not both Boolean, the value `:error:` is pushed onto the stack. In the case that an error results, any values popped off the stack must be pushed back on the stack, in the same order, before `:error:` is pushed onto the stack.
    - The operator `not` is unary and therefore consumes the top value from the stack, pushing the result (a single value) onto the stack. If there isn't a Boolean value on the stack (i.e. the stack is empty or a non-Boolean is on the stack) the value `:error:` is pushed onto the stack. In the case that an error results, any value popped off the stack must be pushed back on before `:error:` is pushed onto the stack.
- Applications of primitive relational operators. Only these primitive relational operators are supported: `equal` and `lessThan`. The latter refers to numerical less than ordering (i.e. there is no support for ordering of non-numeric types).
    - The primitive operators `equal` and `lessThan` are binary and therefore consume the top two values from the stack, pushing the result (a single value) onto the stack[4]. If there are not two values on the stack, or if the two values on the stack are not both numbers, the value `:error:` is pushed onto the stack. In the case that an error results, any values popped off the stack must be pushed back on the stack, in the same order, before `:error:` is pushed onto the stack.
    - The primitive operator `equal` is defined for values of all types except closures: `x y equal` must push `:true:` if `x` and `y` are the same value, `:false:` otherwise. If either x or y is a closure the result must be `:false:`
- `bind` binds a name to a value. The general form is `e`$_2$ `e`$_1$ `bind`
    - `bind` is evaluated by popping two values from the stack. The first value popped, `e`$_1$, can be any expression, including a value pair. The second value popped, `e`$_2$, must be a name or a value pair.
    - `If e`$_1$ is a value pair bind uses the *not-only-local* value.
    - `If e`$_2$ is a value pair bind uses the *local-only* value. If it is a name, binding proceeds. If it is not a name, an error results.
    - Assuming that `e`$_2$ resolves to a name, bind verifies that it is unbound in the current environment.
    - If the name is already bound in the current environment, an error occurs.
    - If the name is unbound in the current environment the name-value binding is added to the current environment. The value that the name was bound to is pushed onto the stack.
    - If there are not two values on the stack, the value `:error:` is pushed onto the stack. In the case that an error results, any values popped off the stack must be pushed back on the stack, in the same order, before `:error:` is pushed onto the stack.
- `load` is evaluated by popping one value from the stack, which must be a string denoting a path, absolute or relative, to a file. The REPL evaluates expressions from the file popping and pushing results as indicated by these rules. The value `:true:` is pushed onto the stack if the contents of the file loads properly (even if the instructions in the file push :error: to

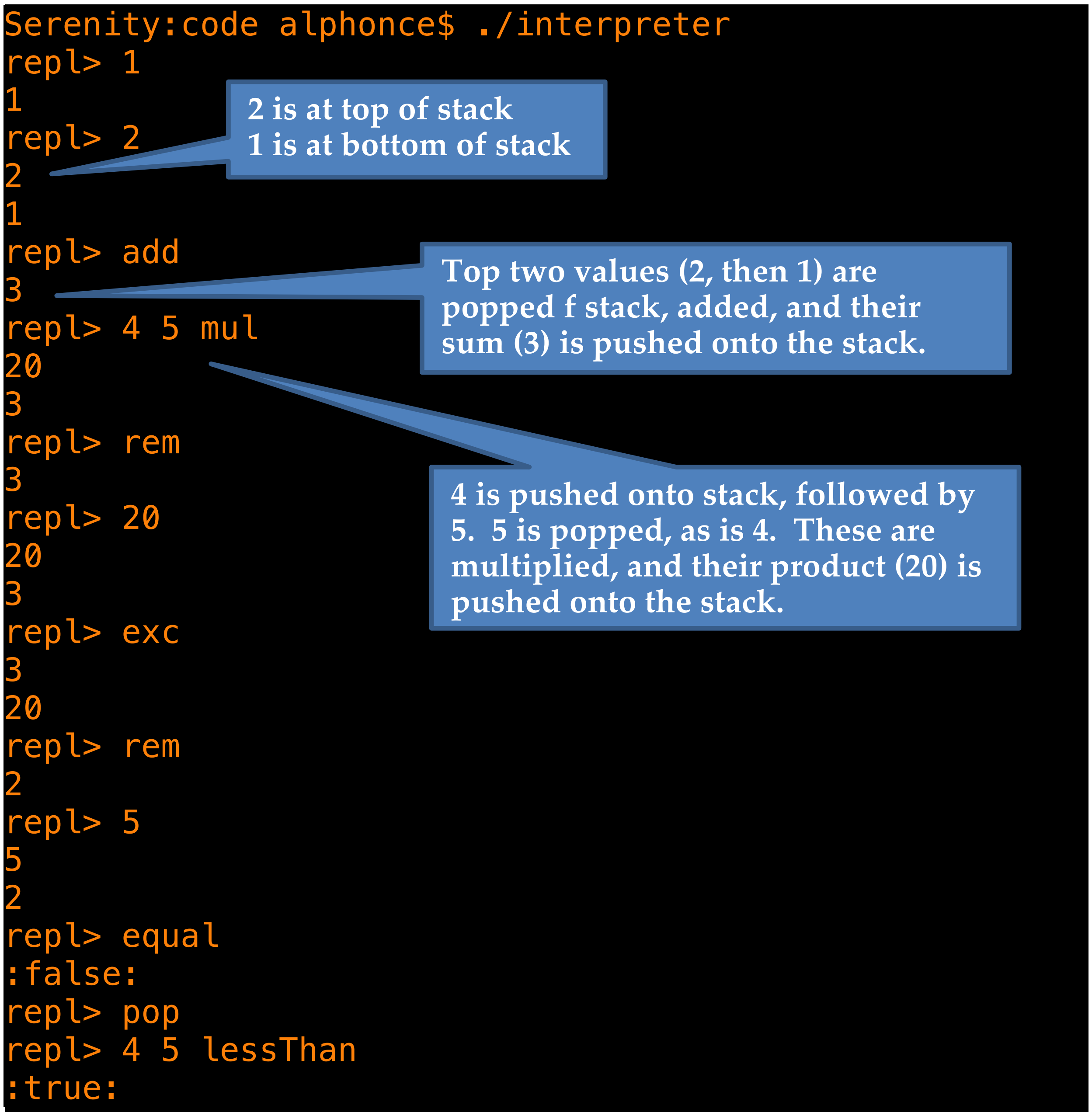
---

[4] `x y lessThan` computes $x < y$.

the stack); if the the file does not exist, `:false:` is pushed onto the stack.

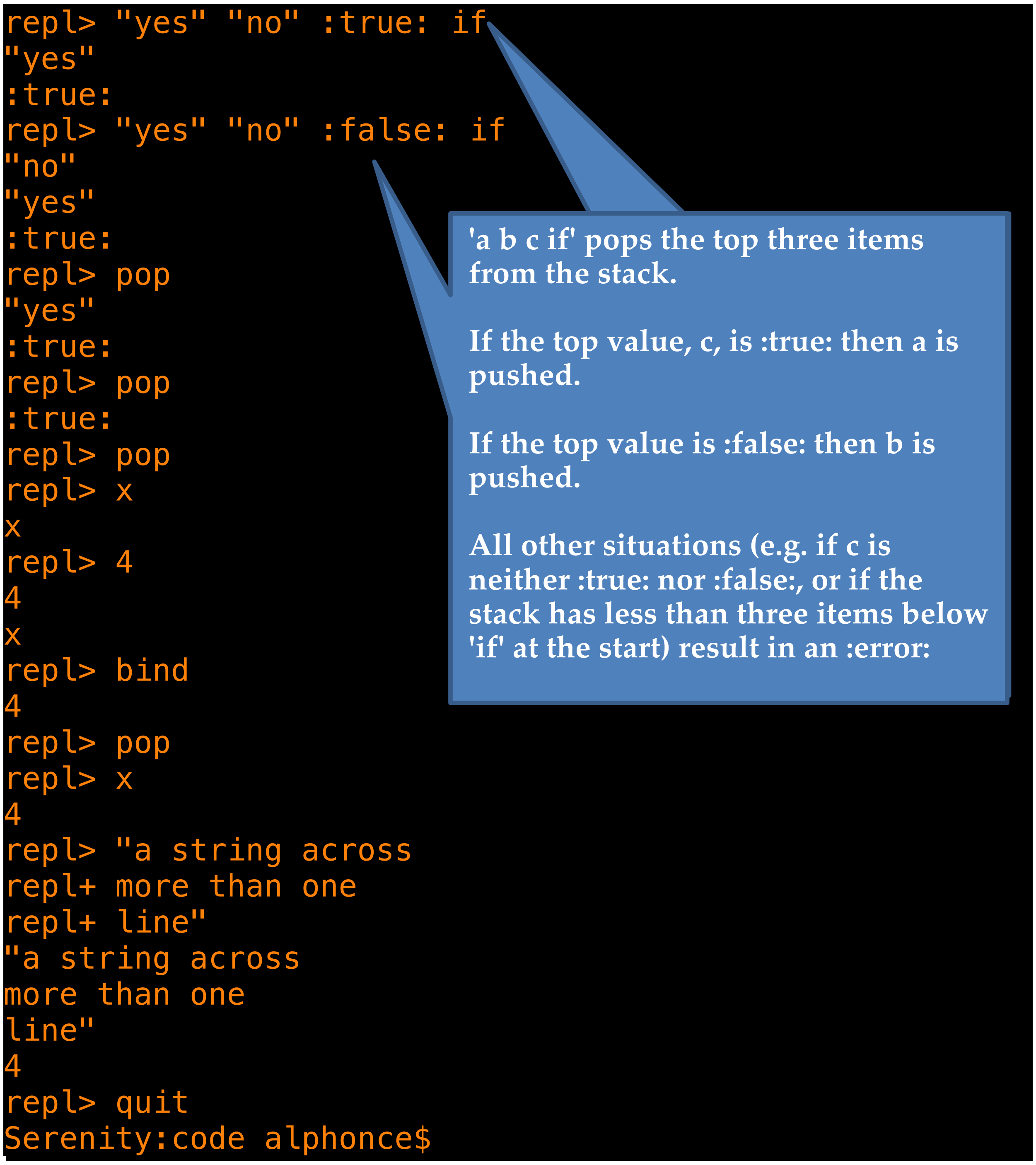
- `if` pops three values off the stack; `x y :true: if` has value `x` whereas `x y :false if` has value `y`. An `:error:` results from any other stack configuration with `if`.
- `pop` removes the top element from the stack. If the stack is empty an error occurs and `:error:` is pushed onto the stack.
- `exc` interchanges the top two values on the stack. If the stack is empty or there is only one item on the stack an error occurs and `:error:` is pushed onto the stack.
- `quit` exits the repl.

Interaction example



```
Serenity:code alphonce$ ./interpreter
repl> 1
1
repl> 2
2
1
repl> add
3
repl> 4 5 mul
20
3
repl> rem
3
repl> 20
20
3
repl> exc
3
20
repl> rem
2
repl> 5
5
2
repl> equal
:false:
repl> pop
repl> 4 5 lessThan
:true:
```

2 is at top of stack
1 is at bottom of stack
Top two values (2, then 1) are popped f stack, added, and their sum (3) is pushed onto the stack.
4 is pushed onto stack, followed by 5. 5 is popped, as is 4. These are multiplied, and their product (20) is pushed onto the stack.



```
repl> "yes" "no" :true: if
"yes"
:true:
repl> "yes" "no" :false: if
"no"
"yes"
:true:
repl> pop
"yes"
:true:
repl> pop
:true:
repl> pop
repl> x
x
repl> 4
4
x
repl> bind
4
repl> pop
repl> x
4
repl> "a string across
repl+ more than one
repl+ line"
"a string across
more than one
line"
4
repl> quit
Serenity:code alphonce$
```

'a b c if' pops the top three items from the stack.

If the top value, c, is :true: then a is pushed.

If the top value is :false: then b is pushed.

All other situations (e.g. if c is neither :true: nor :false:, or if the stack has less than three items below 'if' at the start) result in an :error:

Bug reports

A. x y sub should produce x-y, but instead produces y-x
B. x 0 div should produce :error: but instead crashes the interpreter
C. x 0 rem should produce :error: but instead crashes the interpreter
D. 1 2 x div should result in the follow stack,

```
:error:
x
2
1
```

but instead it is

```
:error:
2
x
1
```

E. String values are intermittently not entered correctly. For example:

```
repl> "a"
"a"
repl> "a"
"a"
"a"
repl> "a"
"aa"
"a"
"a"
```

Notice that "a" showed up in the stack once as "aa". This bug is intermittent, so will not consistently show up.

Feature requests

I. Add a primitive list type, with two type constructors: `[]` (the empty list) and `prepend` (aka cons). Define built-in functions `first` and `rest` which push :error: if applied to anything other than a non-empty list (re-pushing any arguments popped off the stack), but which otherwise return the first member of a list or the rest of the list.
II. Make sure an empty list prints as `[]` and a non-empty list using typical list notation, as in `[1 2 3 4]` for the list `[] 4 prepend 3 prepend 2 prepend 1 prepend`.
III. Add string primitives for `length` (the number of characters in a string) and `concat` (builds a new string consisting of the concatenation of the characters of two existing strings). For example: `“” length` must push 0, `“River” length` must push 5, `“Good” “ morning!” concat` must push `“Good morning!”`.

Expectations

We expect you and your teammates to demonstrate:

1) use of the tools we have discussed to date:
    a. gcc/clang (e.g. try out both compilers if you run into errors to possibly get different error messages)
    b. make (e.g. use make to compile the project)
    c. your favorite editor on timberlake (e.g. emacs)
    d. git (e.g. branch, merge, commit, and push frequently; make sure that each team member pushes their code themselves, so that the git logs reflect the commits of each team member)
    e. gdb (e.g. explore what is happening in memory to track down bugs, especially the intermittent string bug)
    f. Criterion (e.g. write tests which first set up the correct execution environment, then parse and/or evaluate test input)
2) an ability to work effectively as a team,
3) application of the "golden rules",
4) a willingness to ask questions, in office hours and in Piazza.

You will document your process in part through your git logs, but also commit to your repo (e.g. to each bugfix branch or feature development branch) documents detailing your plan/progress to tackle a given task, showcasing key debugger or testing output, etc. You can also embed your thoughts in comments (e.g. documenting your rationale for each Criterion test written).

Take point (4) above seriously. We expect that there will be things with which you are not be familiar (e.g. some C language features used in the code, or how to incorporate needed changes in the makefile). We know that the α language is new to you and your teammates. Make a good faith effort to sort things out on your own, but do not let too much time pass by if fairly progress is not being made. Imagine the scenario that this is your job. Don't go to your manager with questions you **should** be able to answer on your own, but don't be unproductive because you're waiting too long to reach out for a nudge in the right direction.