

Programming Language

Purpose Develop a language which has more types.

Exercises

This homework extends the BAE language from your last assignment. As we will be adding functions, the language will now be called BARF (short for Booleans and Real Functions)!

Exercise 1 Change all references from BAE to BARF.

Exercise 2 Incorporate feedback given on the last assignment, if any.

The syntax of the language is going to get an overhaul. Here is an example of a program in BARF:

```
{program
  {fun even? {n}
    {if {= 0 n} True {call odd? {- n 1}}}}
  {fun odd? {n}
    {if {= 0 n} False {call even? {- n 1}}}}
  {fun main {n}
    {if {= n 1}
      1
      {+ 1 {call main
          {if {call even? n}
            {/ n 2}
            {+ 1 {* n 3}}}}}}}}}
```

The syntax of a program is now:

```
<PROGRAM> ::= { program <FUN> ... }
<FUN>      ::= { fun <id> { <id> } <BARF> }
<BARF>     ::= <same as before> | {call <id> <BARF> }
```

Note that this means (for now), functions are not first-class values (and in particular, this language does not have anonymous functions).

Exercise 3 Define as many types as there are non-terminals in this grammar.

Exercise 4 Define a parser (which takes in a SExpr) for each type of non-terminal.

Exercise 5 Redefine your function which takes in a String to call the correct top-level parser.

Exercise 6 Clearly, your evaluator will need to be able to lookup a function given a function name. Define a function which takes in the name of a function and a PROGRAM and returns the FUN named this. For simplicity, you can assume that there are no duplicate function names. Use `ormap` and `and` in your definition for this function. Read how `ormap` and `and` work in Racket in the docs; they do not have to return booleans! This function should error if no such function is found, and do not do any duplicate computation! **Hint:** You can define a local variable with `let` or `define` to avoid duplicate computation, or just use `or`!

Exercise 7 Running a program now means running the `main` function of the program, whatever that may be, on some input (which must be a number or boolean). Update your function which runs a program to reflect this change. In addition, you will also need to change the signature of `eval` to keep track of a PROGRAM, as it will always need to lookup a function. **Hint:** Running a program is now equivalent to evaluating the `main` function being called on an input. Construct an appropriate BARF to evaluate in your function which runs a program, and evaluate it.

Exercise 8 Now, the body of `eval` must be updated. It would be cumbersome to repeatedly pass the PROGRAM to every `eval-number`, `eval-boolean`, and `eval-function` call, so wrap the body of your `eval` function in the following `let` expression:

```
(let ([eval (lambda ([e : BARF]) <fill-in>)]
      [eval-number <fill-in>]
      [eval-boolean <fill-in>]))
```

where each locally defined function calls the correct top-level function with the whole PROGRAM.

Exercise 9 Now function calls must be evaluated, which is what really separates BARF from BAE. Recall from class how evaluating function bodies is very similar to evaluating with expressions, and proceed accordingly!

Exercise 10 None of your old tests work anymore... sad! Fix this by rewriting them in terms of run*:

```
(: run* : String -> (U Number Boolean))
; a version for testing simple BARF expressions without
; function calls
(define (run* str)
  (eval (parse-expr (string->sexpr str)) (Funs null)))
```

Exercise 11 Write thorough tests for your new version of run!