# Notes and Tips, week 6

Subscribe for email updates. PINNED

No tags yet. + Add Tag

Sort replies by: Oldest first Newest first Most popular

Charilaos Skiadas COMMUNITY TA · a month ago

## Notes on material

- Can "simulate" constructors via lists, where the first element (car) specifies the "constructor".
- Write "symbols" in Racket by starting with a quote, then characters. Eg `'foo`.
- Symbols have fast comparison via `eq?`.
- `(struct foo (bar baz) #:transparent)` creates a new "struct" `foo` with "fields" `bar` and `baz`. This adds a number of functions to the environment:
  - `(foo e1 e2)`
  - `(foo? e)`
  - `(foo-bar e)`
  - `(foo-baz e)`
- structs are not lists!
- Interpreters evaluate a program, Compilers translate it to another language
- We write language B programs by writing their abstract syntax tree directly into the meta-language using constructors/structs.
- Can assume AST is made entirely of language B constructs. But must check the type of those constructs when you evaluate subexpressions.
- Need to be evaluating in an environment, so that variables can be looked up somewhere.
- Need to properly pass an environment to subexpressions (often the same as the one you were given, but not always!)
- "Macros" are basically meta-language (Racket) functions, that take in "language B" expressions and produce "language B" expressions

## Notes on assignment

- Don't forget to submit IN TWO PLACES! (normal, and the peer assessment)
- You can assume your programs are "legal ASTs". Make sure you understand what that means (https://class.coursera.org/proglang-003/lecture/277). This is very important. There are some things your code needs to check for, and other things it can assume.
- Do NOT change the `envlookup` function. But do make sure to use it!
- Two cases are already given to you. Do NOT change them, add the remaining ones.
- There is a final catchall case in the cond, do NOT change that either. Add your cases before it.
- Even though closures should not appear in user programs, so in theory your `eval-exp-env`

should not need to handle them, the autograder does expect it to. So you should make sure that your `eval-exp-env` has a `closure?` case, and remember that closures are values.

- If you evaluate a pair expression, and it happens to be a "value" already (i.e. its components are values already), then you may if you wish return that pair itself instead of creating a new one (If this comment does not make sense to you, then you probably don't need to worry about it).
- The environment is just a Racket list, extend it using `cons` when you need to.
- Make sure to write a lot of tests for your proper handling of lexical scope!
- Make sure to account for the possibility of anonymous functions
- Make sure to account for shadowing of variables
- For question 3, heed the warning about not using `eval-exp` and/or closures. Your Racket functions should simply produce a MUPL expression, that *when evaluated would do the right thing*.
- You do NOT need to use `racketlist->mupllist` for `mlet*`.
- For `ifeq`, make sure that when your resulting MUPL expression is evaluated, the expressions `e1` and `e2` are evaluated exactly once each.
- The most important thing in this assignment, and hardest part to get used to, is being able to clearly identify what parts are MUPL expressions, and parts are Racket constructs, and when to evaluate or not.
  - For instance, you define "MUPL macros" by writing Racket functions whose arguments are assumed to be MUPL expressions, and whose body/result needs to also be a (unevaluated) MUPL expression that uses these arguments. If that expression were to be evaluated via `eval-exp`, it should have the effect described for the macro. But that evaluation is not part of the macro definition.
  - Similarly, to define a "MUPL function `f`", you would basically create a Racket binding `(define mupl-f e)` where `e` is a MUPL expression of the form `(fun ...)` that when evaluated would result in a MUPL function with the required properties. You would normally then use this function in a MUPL program via an assignment `(mlet "f" mupl-f ...)`. The assignment's `mupl-mapAddN` has an example of using `mupl-map` that way.

13 · flag

Jonas Collberg · a month ago

No hints on the challenge problem this week? ;)

1 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Hehe well that's probably partly because of how much time I spent preparing the extra problems for this week.

But I don't know, this week's challenge problem seems somewhat straightforward to me, but maybe I've spent too much time with that stuff. You really just need an inner helper in compute-free-vars that takes in an expression and returns a pair of the modified expression as well as a set of the free variables in the expression. Implementing it would be a long case expression to recurse into subexpressions, get their modified forms and sets of free

variables, and create your response from that, so a somewhat standard recursive function.

Other than that, your new eval needs a more careful treatment of the fun-challenge branch, as you need to bottle up in the closure only the bindings of the free variables, and then a revision of the call branch, as well as making sure that all recursive calls to eval-under-env have now switched to being eval-under-env-c. You don't want to suddenly jump to the non-challenge eval just because you forgot to change the `apair?` recursive calls.

3 · flag

Jonas Collberg · a month ago

That was exactly what I needed, I think (hope?). Thanks a lot!

0 · flag

Dominique · a month ago

In your last paragraph, you bind a function to a variable in `mlet`.

A program that uses such a binding would have to apply '(eval (var mupl-f))' to extract the closure and pass it to `call`.

So I understand the use of `eval` is not restricted to run the program.

0 · flag

Marco Fabbri Signature Track · a month ago

Dominique, if you bound the mupl-f function to the MUPL variable "f" (so variable f points to a closure, and a clousure evaluates to itself) you can actually call it with:

```
(call (var "f")  (int 42))
```

2 · flag

Laura Dean · a month ago

Do `compute-free-vars` and `eval-under-env-c` need to handle to oddball case of getting a `closure` passed in? If so, would compute-free-vars receive one containing a `fun` or a `fun-challenge`? (And could it safely assume that the `closure-env` needs no adjustment?)

1 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Technically closures are already values, so you shouldn't really need to be doing anything

with them. My code just returns them and seemed to work just fine. You should definitely assume that they are based off fun-challenge and don't need adjustment.

1 · flag

A post was deleted

Ian Perkins · a month ago

Just to follow up on the `(call (var "f") (int 42))` post.

It seems that `(call mupl-f (int 42))` works just as well *if* mupl-f is a Racket variable bound to a MUPL `fun`. In that case, the Racket variable gets (pre?)-evaluated to be a MUPL expression so `eval-exp` will then create a closure from it and everything will just work.

I'm not 100% sure how this works and I can see that it only works for the case mentioned above. Hopefully the auto-grader will not have an issue with it as it makes my `mupl-mapAddN` solution 1 line shorter (although I do now ignore the provided `mlet`)

***update*** posted this and then realised that, yes, of course it works - it is a macro duh. Anyway, left this comment here in case it helps anyone...

p.s. any chance of switching off the spell-checker? - it's taken me 3 attempts to write `mlet` !!!

0 · flag

Aditya Athalye Signature Track · 25 days ago

Just sayin'... I *still* haven't figured out how to write compute-free-vars.
Last few hours, some more attempts before the hard deadline...

0 · flag

Peter Eriksen COMMUNITY TA · 25 days ago

@Aditya Athalye: Do you get what the type of `compute-free-vars` is supposed to have? Which cases have you implemented, and which of them look like they are working fine?

0 · flag

Andrés Ferrari · 25 days ago

@Aditya If it's any consolation, I couldn't make all the challenge tests pass. I just submitted whatever I had for the challenge; I'll probably try to achieve 105 points after the hard deadline (I'm stubborn!). Currently one of the tests for `compute-free-vars` fails (the one about "no free variables" or something like that). I think I have a bug in my handling of either `mlet` or `fun`, or the interplay between those two.

0

· flag

Aditya Athalye Signature Track · 25 days ago

Houston, we have liftoff!
(Maybe :-) )

@Charilaos,
Your comment gave me a lot of confidence that I was thinking in the right ballpark. You are right; after developing the correct intuition, the implementation of *compute-free-vars* is almost boring. Thank you!

@Peter,
at the time of my previous comment, I understood exactly what *compute-free-vars* was supposed to do, but I had net *zero* working cases.
That was because I had not realised how to separate out the job of transforming expressions v/s the job of computing freevars. Now I think I have a correct intuition about the solution as well.

@Andrés,
I believe I have a working solution... just (nervously) going over things before I submit to the autograder...
I have at least 10 solid hours of thinking and trial-and error behind my current solution. What really works for me every time is to shut my computer, take a pen and paper and evaluate progressively more sophisticated test cases by hand.
The eval-on-paper process really helps me "forget" specific implementation details, and just develop an intuition for the behaviour I should expect.

Thank you all for your time and comments.

Here is one of my test cases and expected result:

```
(check-equal?
 (eval-exp-c
  (mlet* (list (cons "y" (int 5))
               (cons "z" (int 5)))
         (call
          (fun #f "x"
               (call
                (fun "quxx" "moo"
                     (add (add (call (fun #f "foo"
                                          (add (var "x") (var "y")))
                                     (int 5))
                               (call (fun "bar" "baz"
```

```
                                        (add (var "baz") (var "y")))
                                    (int 5)))
                            (var "z")))
                    (int 5)))
            (int 5))))
  (int 25))
```

0 · flag

Aditya Athalye Signature Track · 25 days ago

101.00 / 100.00
Feedback
compute-free-vars: correctly computes free vars [incorrect answer]
compute-free-vars: no free vars case [incorrect answer]

I suppose that means eval-under-env-c is correct, and I still need to work on compute-free-vars. Anyhow, really happy that I at least have a partially working solution :)

0 · flag

Andrés Ferrari · 25 days ago

Not bad! I wrote lots of tests for `compute-free-vars` and `eval-under-env-c`, but I still fail:

```
compute-free-vars: no free vars case [incorrect answer]
```

This leads me to believe I've probably misunderstood what they are supposed to do...

0 · flag

+ Comment

Rafael La Buonora · a month ago

Should we account for shadowing? I think envlookup already works with shadowing, since it gets the last item added with that name:

```
(define env1 (list (cons "one" (int 1)) (cons "rafa" (int 30))))
(envlookup env1 "one"); -> 1
(envlookup (cons (cons "one" (int 9)) env1) "one") ; -> 9
```

Isn't that enough?

0 · flag

Chad Miller · a month ago

Yes, outside of the challenge problem this works fine as long as you're building `env` correctly.

0 · flag

+ Comment

Ezra Schroeder · a month ago

When should we use envlookup? I am almost done with the homework (was thinking and hoping maybe I *was* done with it but my call? branch of the conditional *crashes* on everything I give it to play with). I didn't use envlookup anywhere. I haven't tried the challenge part yet. Everything I have thought of to test my solution to the homework seems to work, except if I try to call the 'call' branch the program crashes with errors.

0 · flag

Chad Miller · a month ago

only `var` should directly use envlookup. The catch, though, is that any time `eval-exp` encounters anything could be a `var`, you need to `eval-exp` it (note: don't use `if`s to decide if something is a var or not; if something is a non-var value it'll just eval to itself anyway). Failing to do so can get errors like you describe, and `call` is an easy place for these kinds of mysterious errors.

e.g. consider:

```
(apair (int 1) (int 2))
```

It's possible for this to evaluate fine and make you think your apair code is correct, but then try:

```
(mlet "x" (int 1)
    (apair (var "x") (int 2)))
```

And suddenly it breaks if you're not `eval-exp`'ing the subexpressions like you need to. Similar things that can break:

```
(fst (apair (int 1) (int 2)) ; may work even if
(mlet "p" (apair (int 1) (int 2)) (fst (var "p"))) ; doesn't
```

```
(call (closure null (fun #f "x" (var "x"))) (int 1))
; vs
(mlet "f" (closure null (fun #f "x" (var "x")))
     (call (var "f") (int 1)))
```

1 · flag

Ezra Schroeder · a month ago

Thanks Chad!,

I continue to have difficulties (for example when I run your provided ideas some work some don't):

> (mlet "x" 1 (apair (var "x") (int 2)))
(mlet "x" 1 (apair (var "x") (int 2)))
> (eval-exp (mlet "x" 1 (apair (var "x") (int 2))))
. . bad MUPL expression: 1
>

> (fst (apair (int 1) (int 2)))
(fst (apair (int 1) (int 2)))
> (eval-exp (fst (apair (int 1) (int 2))))
(int 1)
>

> (mlet "p" (apair (int 1) (int 2)) (fst (var "p")))
(mlet "p" (apair (int 1) (int 2)) (fst (var "p")))
> (eval-exp (mlet "p" (apair (int 1) (int 2)) (fst (var "p"))))
(int 1)
>

> (call (closure null (fun #f "x" (var "x"))) 1)
(call (closure '() (fun #f "x" (var "x"))) 1)
> (eval-exp (call (closure null (fun #f "x" (var "x"))) 1))
. . bad MUPL expression: 1
>

> (mlet "f" (closure null (fun #f "x" (var "x"))) (call (var "f") 1))
(mlet "f" (closure '() (fun #f "x" (var "x"))) (call (var "f") 1))
> (eval-exp (mlet "f" (closure null (fun #f "x" (var "x")))
    (call (var "f") 1)))
. . bad MUPL expression: 1

>

In the hw5test.rkt file that was provided (or whatever it's called) if I comment out anything that *calls*
anything, I get something about "9 of 9 tests passed, 0 failures". If I try to call *anything* my code breaks.
Typically with "bad MUPL expression: <whatever>".

Any thoughts?

0 · flag

Ezra Schroeder · a month ago

Also,

I think my mupl-mapAddN is functional:

```
> (define int-zs (apair (int 1) (apair (int 3) (apair (int 5) (aunit)))))
> int-zs
(apair (int 1) (apair (int 3) (apair (int 5) (aunit))))
> ((mupl-mapAddN (int 7)) int-zs)
(apair (int 8) (apair (int 10) (apair (int 12) (aunit))))
> (eval-exp ((mupl-mapAddN (int 7)) int-zs))
(apair (int 8) (apair (int 10) (apair (int 12) (aunit))))
>
```

0 · flag

Ezra Schroeder · a month ago

Even the provided "call" test that doesn't involve mupl-mapAddN is not working for me:

```
> (call (closure '() (fun #f "x" (add (var "x") (int 7)))) (int 1))
(call (closure '() (fun #f "x" (add (var "x") (int 7)))) (int 1))
>  (eval-exp (call (closure '() (fun #f "x" (add (var "x") (int 7)))) (int 1)))
. . application: not a procedure;
 expected a procedure that can be applied to arguments
  given: (cons "x" (int 1))
  arguments...: [none]
>
```

0 · flag

Ezra Schroeder · a month ago

One mistake that I think I have is that I commented out the "suggested" (I thought it was suggested at the time) beginning to mupl-mapAddN so I could have a curried function

```
(define mupl-mapAddN
;;  (mlet "map" mupl-map
      (λ (<something>)
        (λ (<something else>) ...
```

I really have no idea how I could use the provided "suggested" beginning to mupl-mappAddN if I uncomment it out, and I also don't know how I would get back the needed curried behavior of mupl-mapAddN if I do.

Another *mistake* (?) is that in the middle of my mlet? branch of my conditional in the eval-function, at one point I have something that looks like

```
(append (list (cons ...
```

which seems like it has to be a mistake (but I can run mlet and mlet* on all sorts of different stuff, seemingly correctly...).

0 · flag

Chad Miller · a month ago

I made a mistake in those examples; everywhere it said `1` was supposed to be `(int 1)`. Sorry. I just edited it.

The last error looks like a bug in your own code, however; namely it looks like there's a `cons` where there should be an `apair`.

Here are some tests for Problem 2 that I have run and tested:

```
(check-equal? (eval-exp (int 17)) (int 17) "(eval-exp (int 17))")
(check-equal? (eval-under-env (var "x") (list (cons "x" (int 17)))) (int 17) "
single-variable lookup")
(check-equal? (eval-exp (add (int 1) (int 2))) (int 3) "adding two ints")
(check-equal? (eval-exp (ifgreater (int 1) (int 0) (int 1) (int 0))) (int 1) "
ifgreater: true case")
(check-equal? (eval-exp (ifgreater (int 0) (int 0) (int 1) (int 0))) (int 0) "
ifgreater: false case")
(check-equal? (eval-exp (apair (add (int 1) (int 2)) (ifgreater (int 2) (int 1
) (int 3) (int 0)))) (apair (int 3) (int 3)) "apair of adds")
(check-equal? (eval-exp (aunit)) (aunit) "eval-exp aunit")
(check-equal? (eval-exp (isaunit (aunit))) (int 1) "isaunit: true case")
(check-equal? (eval-exp (isaunit (int 100))) (int 0) "isaunit: false case")
```

```
(check-equal? (eval-exp (mlet "xs" (apair (int 1) (int 2)) (var "xs"))) (apair
 (int 1) (int 2)) "mlet with pair")
(check-equal? (eval-exp (mlet "xs" (apair (int 1) (int 2)) (fst (var "xs"))))
(int 1) "mlet + fst")
(check-equal? (eval-exp (fun #f "x" (aunit))) (closure null (fun #f "x" (aunit
))) "fun evaluates to closure")
(check-equal? (eval-exp (call (fun #f "x" (add (var "x") (int 1))) (int 1))) (
int 2) "call a fun struct")
```

1 · flag

Chad Miller · a month ago

The mlet is just there so you can write `(call mupl-map)` in your solution.

1 · flag

Ezra Schroeder · a month ago

My code passes everything *except* the last one

(eval-exp (call (fun #f "x" (add (var "x") (int 1))) (int 1)))
. . application: not a procedure;
 expected a procedure that can be applied to arguments
  given: (cons "x" (int 1))
  arguments...: [none]
>

0 · flag

Ezra Schroeder · a month ago

Actually, my code also failed the first (aunit) test (gave (int 0) when it should have said (int 1), but I suspect it's a logic error which I'll fix momentarily, and I suspect it isn't hurting anything else anywhere.

0 · flag

+ Comment

Rafael La Buonora · a month ago

My call works but not on recursive functions. When I evaluate a call to a recursive function I get it's body as a result. I supposed it's something about Charilaos's last tip but ...

0 · flag

Chad Miller · a month ago

almost certainly a problem with not correctly including the function name in the `env`.

0 · flag

+ Comment

Erik Colban · a month ago

Since `closure`s are not part of "source programs", can we assume that they are syntactically correct? Or do we need to test that `(closure-fun a-closure)` is really a `fun?`? Likewise, can we assume that the environment of a closure is a list of (string, value) pairs?

0 · flag

Chad Miller · a month ago

We're allowed to "fail mysteriously" on bad syntax. So no, don't bother.

0 · flag

Erik Colban · a month ago

The thing is, it's a MUPL type error, something we should generate an error for. If the following code were part of MUPL syntax,

```
(closure <env> <funexpr>)
```

then my understanding of what we may and may not assume, is that we may not assume that `<funexpr>` indeed is a `fun?`, and that we would need to check for it and, if not a `fun?`, generate an error with a MUPL specific error. So syntax like

```
(eval-exp (call (closure '() (aunit)) (int 1)))
```

should generate a MUPL error.

0 · flag

Chad Miller · a month ago

Well, as you said, a `closure` isn't ever supposed to be created by the end-user, so such an invalid `closure` can't ever be created by MUPL code. And the only legitimate reason for `eval-under-exp` to create a `closure` is if it evaluates a `fun`. So it's one of those "can't ever happen" things.

Checking for, say, a `call` on a non-callable is a different matter and something you should check for.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

This will generate a MUPL error when the call is evaluated, assuming you've implemented call correctly. It would not generate an error when the closure is evaluated.

0 · flag

Erik Colban · a month ago

So now I am confused. Seems like Charilaos and Chad are saying the opposite. It's the case of `call` that I had in mind. The assignment text hints that the code for this case is no more than 12 lines long, which seems short if we have to test that the `fun` extracted from the extracted `closure` is really a `fun?` and break lines in natural places. For the same reasons that Chad mentioned, I was thinking that there is no need for this extra test.

I think I'll just add the extra test in the two first submissions, then experiment with removing it after the second submission.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Hm sorry, I think I misled you there. You have to test that the first expression in a call is a closure, it would be a MUPL error otherwise. You do not need to test that the fun in the closure is a fun, since you basically are the only one creating the closures (outside of test situations). So there is no setting where you would be given a closure that does not have a fun in it, as you would never generate such a closure.

1 · flag

Erik Colban · a month ago

Thanks, Charilaos. Good to know we are all on the same page.

0 · flag

Ezra Schroeder · a month ago

You guys are truly awesome!

I think in both the (closure? e) and (fun? e) branches of the cond, it's just a one-liner. There's no way to recursively evaluate anything, you just return a closure in both cases.

0 · flag

+ Comment

Anonymous · a month ago

I suspect I have overly complicated my implementation of "call"
....is it easiest to branch on whether (call-funexp e) is a function or a closure?
...in the event that (call-funexp e) is a closure, one might need to do another eval-under-env to get a function out from (closure-fun (call-funexp e), using the (cons (closure-env e) env)? examples
1) a (closure-fun (call-funexp e)) be a variable that needs to be looked up?
2) another closure?

Would love help thinking this through to clean up & make sure I'm understanding what it looks like when big call stacks get built.

0 · flag

Chad Miller · a month ago

You don't need to do any branching on `fun` vs. `closure` because a `fun` evaluates to a closure.

You don't need to reevaluate the `fun` after accessing it from a `closure` because the `fun` object was already evaluated when you created the closure.

2 · flag

Ezra Schroeder · a month ago

In the instructions for (call? e) it says "A call evaluates its first and second subexpressions to values." That uses the *current* environment (i.e. env), right?

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Anything else would be really bizarre behavior.

1 · flag

Anonymous · a month ago

Aha.... I ran into trouble with tests written as follows:

```
(letrec (  [myclosure       (closure '() (fun "myfunction" "x" (apair (var "x") (int 7))))]
           [ myenv          (list (cons "myfunction" myclosure ) )] )
```

(eval-exp (call (closure myenv (var "myfunction")) (int 1)))
)

Grr....

0 · flag

+ Comment

Ezra Schroeder · a month ago

Are people using mupllist->racketlist or racketlist->mupllist in their mupl-maps?

0 · flag

Ezra Schroeder · a month ago

Is it okay to use eval-exp in mupl-map? I know we aren't supposed to use it in #3, but I thought maybe in #4 it's ok.

0 · flag

Chad Miller · a month ago

No to both; they're supposed to be pure MUPL code.

1 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

You can never use eval-exp, since you don't know the environment in which the expression will be evaluated in. In almost all problems, there is nothing you can do at "Racket-time". The only thing you can do is create a MUPL expression that would do the right thing when eval-ed in "MUPL-time".

1 · flag

Ezra Schroeder · a month ago

I thought I might have almost had it, but when I ran the provided test

```
(eval-exp (call (call mupl-map (fun #f "x" (add (var "x") (int 7)))) (apair (
int 1) (aunit))))
```

I got

```
MUPL fst applied to non-apair
```

0 · flag

Ezra Schroeder · a month ago

I wrote this to have an analog to refer to in building mupl-map:

```
(define e-map ;;e-map is the same as the provided map except curried
  (λ (f)
    (λ (xs)
      (if (null? xs)
          null
          (cons (f (car xs)) ((e-map f) (cdr xs)))))))
```

Except I start out my mupl-map like

```
(define mupl-map ;"CHANGE"
  (fun "f" "xs"
```

But then when I get down to the place where my 'e-map' (has nothing to do with mupl-) e-map's f to the cdr of the list, and creating its analog in mupl-map, i'm lost. I can clearly NOT say something like

```
(mupl-map (var "f") (snd (var "xs")))
```

but I don't know what I *can* say...

0 · flag

Ezra Schroeder · a month ago

Here's an example of what it is doing at the moment,

```
> (call mupl-map (fun #f "x" (add (var "x") (int 7))))

(call
```

```
(fun "f" "xs" (ifgreater (isaunit (var "xs")) (int 0) (aunit) (apair (call (var "f") (fst (var "xs"))) (mlet "ys" (snd (var "xs")) (call (var "f") (var "ys"))))))
```

```
(fun #f "x" (add (var "x") (int 7))))
```

```
>
```

But I know it's not functional, the REPL says "***MUPL fst applied to non-apair***", which is my own error message in my

```
(fst? e)
```

in the

```
cond
```

.

0 · flag

Laura Dean · a month ago

Luckily, that's your own error message. :)

What happens if you change your code so that it includes, in that error message, some information about what `fst` is being applied to? (See the `#t` case of the big `cond` for an example of how to do so.) You might then get some additional information that will help you debug things.

2 · flag

Erik Colban · a month ago

Ezra,

I ran into the same problem myself. In my case the error was in my `eval-under-env` implementation. My thinking was that for `(fst e)` and `(snd e)` I would not have to evaluate `e`, but only `(apair-e1 e)`. But that was incorrect. The expression might be `(fst (var "x"))` (that is `e = (var "x")`), which requires the sub-expression `e` to be evaluated first, then to extract (not re-evaluate) the `apair-e1` from the resulting value.

2 · flag

Ezra Schroeder · a month ago

Erik,

you mean in the (fst e?) branch of the cond? I evaluate fst-e e in the current environment. Then, I check if that is apair. If so I extract apair-e1 of it.

0 · flag

Ezra Schroeder · a month ago

Laura,

it used to say "MUPL snd applied to non-apair" (my own error message in the snd branch of the big cond). Now, after putting a format in that error statement, it says "format: format string requires 0 arguments, given 1; arguments were: "MUPL snd applied to non-apair" (closure '() (fun #f "x" (add (var "x") (int 7))))".

That's now what I get when I run the provided MUPL test on my file.

I.e.,

```
(eval-exp (call (call mupl-map (fun #f "x" (add (var "x") (int 7)))) (apair (i
nt 1) (aunit))))
```

Results in

```
format: format string requires 0 arguments, given 1; arguments were: "MUPL snd
 applied to non-apair" (closure '() (fun #f "x" (add (var "x") (int 7))))
```

Which is coming from the error part of my (snd? e) branch of the big cond:

```
(error (format "MUPL snd applied to non-apair" v))
```

Which is in response to taking the false branch of

```
(if (apair? v)
```

where v is bound to the result of evaluating in the current environment (snd-e e)

0 · flag

Ezra Schroeder · a month ago

When I try to run

```
(eval-exp (call (fun "sum" "es"
           (ifaunit (var "es")
                    (int 0)
                    (add (fst (var "es"))
                         (call (var "sum") (snd (var "es"))))))
      (racketlist->mupllist (list (int 10) (int 20) (int 30)))))
```

I get

```
format: format string requires 0 arguments, given 1; arguments were: "MUPL fst
 applied to non-apair" (fst (var "es"))
```

Which is in response to the error part of my (fst? e) of my cond

```
(if (apair? v)
               (apair-e1 v)
               (error (format "MUPL fst applied to non-apair" e))
```

v is the result of evaulating (fst-e e) in the current environment when (fst? e) is reached in the big cond.

0 · flag

Peter Eriksen COMMUNITY TA · a month ago

> `format: format string requires 0 arguments, given 1; arguments were: "MUPL fst applied to non-apair" (fst (var "es"))`

This means that your formatting string doesn't contain any positions to put values in (Like `%d` in C-like languages). Try adding `~v` somewhere in the string like the `bad MUPL expression` error string.

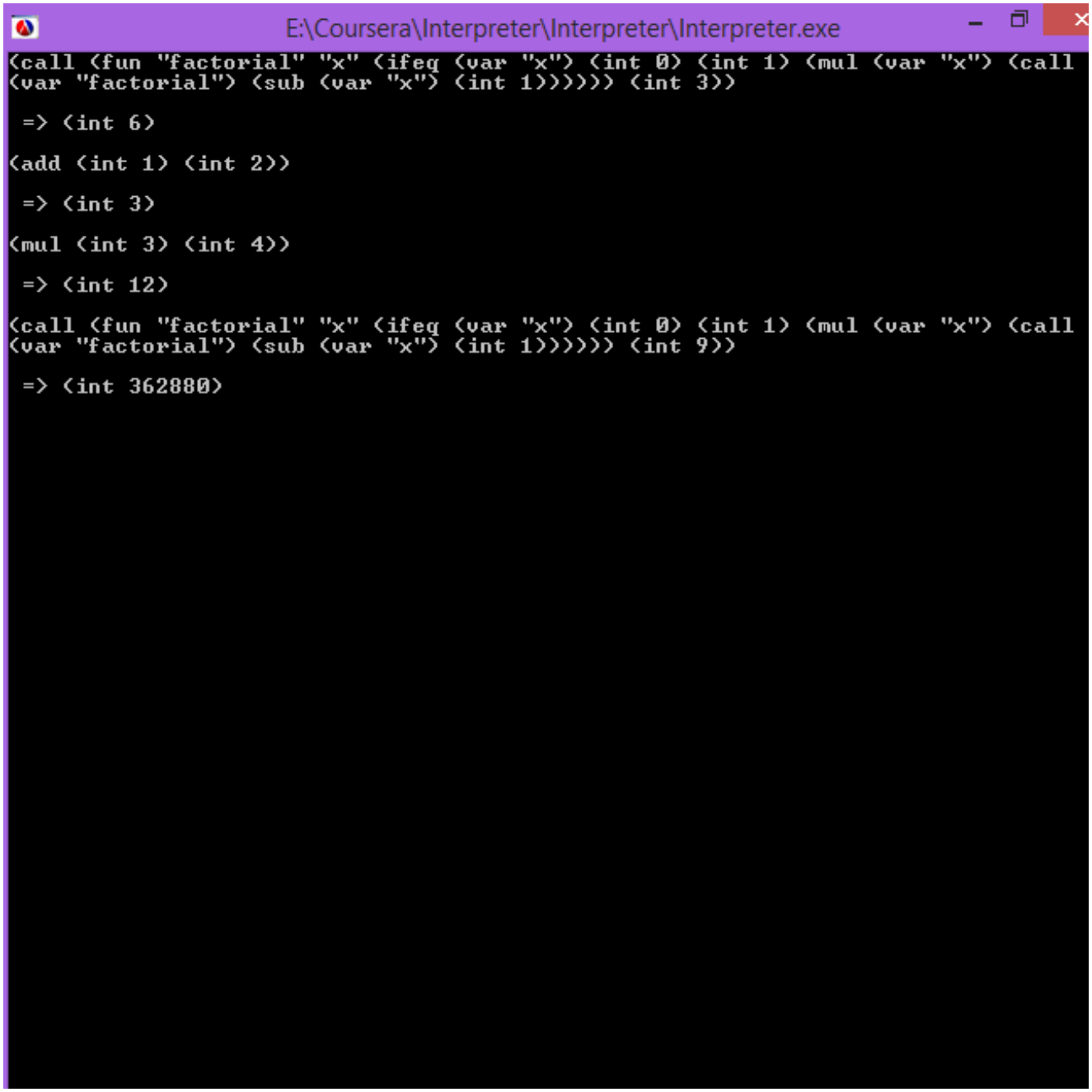
It also says that the variable `es` isn't bound to `apair?`. I think, this is the same problem as you had in another thread, where I also commented.

1 · flag

+ Comment

Anonymous · 23 days ago

Just for fun,I added mul and sub to the MUPL and built a console executable of the interpreter which works like the REPL.




5 · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B | I | Link | <code> | Pic | Math | Edit: Rich | Preview

- [ ] Make this post anonymous to other students
- [x] Subscribe to this thread at the same time

Add post