smol/fun

How to read this file?

```
(a smol/fun program)
```

- **Correct answer**
- **Other answer 1**
- **Other answer 2**

Why the correct answer is correct

arithmetic operators

```
(deffun (f o) (o 1 1))
(f +)
```

- **Error**
- **Syntax error**
- **2**

The correct answer is Error because smol/fun doesn't allow programmers to pass functions as arguments. 2 would have been correct if smol/fun permits higher-order functions, i.e. functions that consume or produce functions, such as f.

0 as condition

```
(if 0 #t #f)
```

- **#t**
- **#f**

In smol/fun, every value other than #f is considered "true". You might find this confusing if you are familiar with Python or C.

redeclare var using defvar

```
(defvar x 0)
(defvar y x)
```

```
(defvar x 2)
x
y
```

- **Error**
- **2; 0**
- **0; 0**
- **Nothing is printed**

You can't redeclare x in the same scope level (the global, in this case).

expose local defvar

```
(defvar x 42)
(deffun (create)
  (defvar y 42)
  y)

(create)
(equal? x y)
```

- **Error**
- **42; #t**

The variable y is declared locally. You can't use it outside of the create function.

pair?

```
(pair? (pair 1 2))
(pair? (ivec 1 2))
(pair? '#(1 2))
(pair? '(1 2))
```

- **#t #t #t #f**
- **#t #f #t #f**
- **#t #t #t #t**

In smol, pair is a special-case of ivec. The last vector-like expression is Racket's way of writing list 1, 2.

let* and let

```
(let* ([v 1]
       [w (+ v 2)]
       [y (* w w)])
  (let ([v 3]
        [y (* v w)])
    y))
```

- **3**
- **9**
- **27**

When the inner y is created with (* v w), the v is the outer v.

defvar and let

```
(defvar x 3)
(defvar y (let ([y 6] [x 5]) x))

(* x y)
```

- **15**
- **25**
- **9**

The global y is defined to be equal to the local x, which is 5.

fun-id equals to arg-id

```
(deffun (f f) f)
(f 5)
```

- **5**
- **Error**

The parameter f shadows the function name f.

scoping rule of let

```
(let ([x 4]
      [y (+ x 10)])
  y)
```

- **Error**
- **14**

The let expression binds x and y simultaneously, so y cannot see x. If you replace let with let*, the program will produce 14.

the right component of ivec

```
(right (ivec 1 2 3))
```

- **Error**
- **2**

The documentation of `right` says that the input must be of type Pair, and an ivec of size 3 can't be a Pair. If the smol/fun does not check that its parameter is a pair, however, this will return 2.

identifiers

```
(defvar x 5)

(deffun (reassign var_name new_val)
  (defvar var_name new_val)
  (pair var_name x))

(reassign x 6)
x
```

- **'#(6 5) 5**
- **'#(6 6) 5**
- **'#(6 6) 6**
- **Error**
- **Nothing is printed**

The inner defvar declare var_name locally, which shadows the parameter var_name. Neither var_name has anything to do with x, which is defined globally.

defvar, deffun, and let

```
(defvar a 1)
(deffun (what-is-a) a)

(let ([a 2])
  (ivec
    (what-is-a)
    a))
```

- **'#(1 2)**
- **'#(2 2)**

The function what-is-a is defined globally. When it uses the variable a, it looks up in the global scope.

syntax pitfall

```
(deffun (f a b) a + b)
(f 5 10)
```

- **10**
- **15**
- **5**
- **Error**

It is easy to forget smol/fun uses prefix parenthetical syntax. To do the right thing, the deffun should be `(deffun (f a b) (+ a b))`. This program produces 10 because when smol/fun computes the value of `(f 5 10)`, it computes a, and computes +, and finally computes b and returns b's value, which is 10.