A RACKET-PYRET: SURVEY 1

To summarize, Survey 1 presented material in the following order:

(1) Survey explanation: Figure 6.
(2) Pyret programs: Figure 7. These were shown before the Racket programs to avoid priming them.
(3) One sentence signaling the transition to Racket programs: "Next, we are going to show you several Racket programs and ask you to tell us what they produce."
(4) Racket programs: Figure 8.
(5) The prompt: "Finally, we are going to ask you to compare programs across the two languages that you have seen."
(6) All program pairs: Figure 9.
(7) Questionnaire on student background: Figure 10.

All the programs are in Table 2.

To prepare for the semester, we are going to show you several programs and ask you to tell us what they produce.

Some programs are written in Racket, while others are written in a hypothetical language. Some programs have multiple expressions; in that case, each one has its own response area.

We ask you to answer these *without running* the programs. It's okay if you're wrong! Any incorrectness in your responses will **NOT** affect your placement status, grade, or anything else. We only ask you to treat these questions seriously. They are an important part of your learning process, and will also affect learning for future students. We will explain how this connects to the course after the deadline.

You are welcome to write the same answer for multiple questions.

We estimate that the entire survey will take you about 2 hours. You are welcome to stop at any time and resume later; the system will keep track of where you were.

Please complete the survey by August 19.

Fig. 6. Racket-Pyret: Survey 1: Opening Page

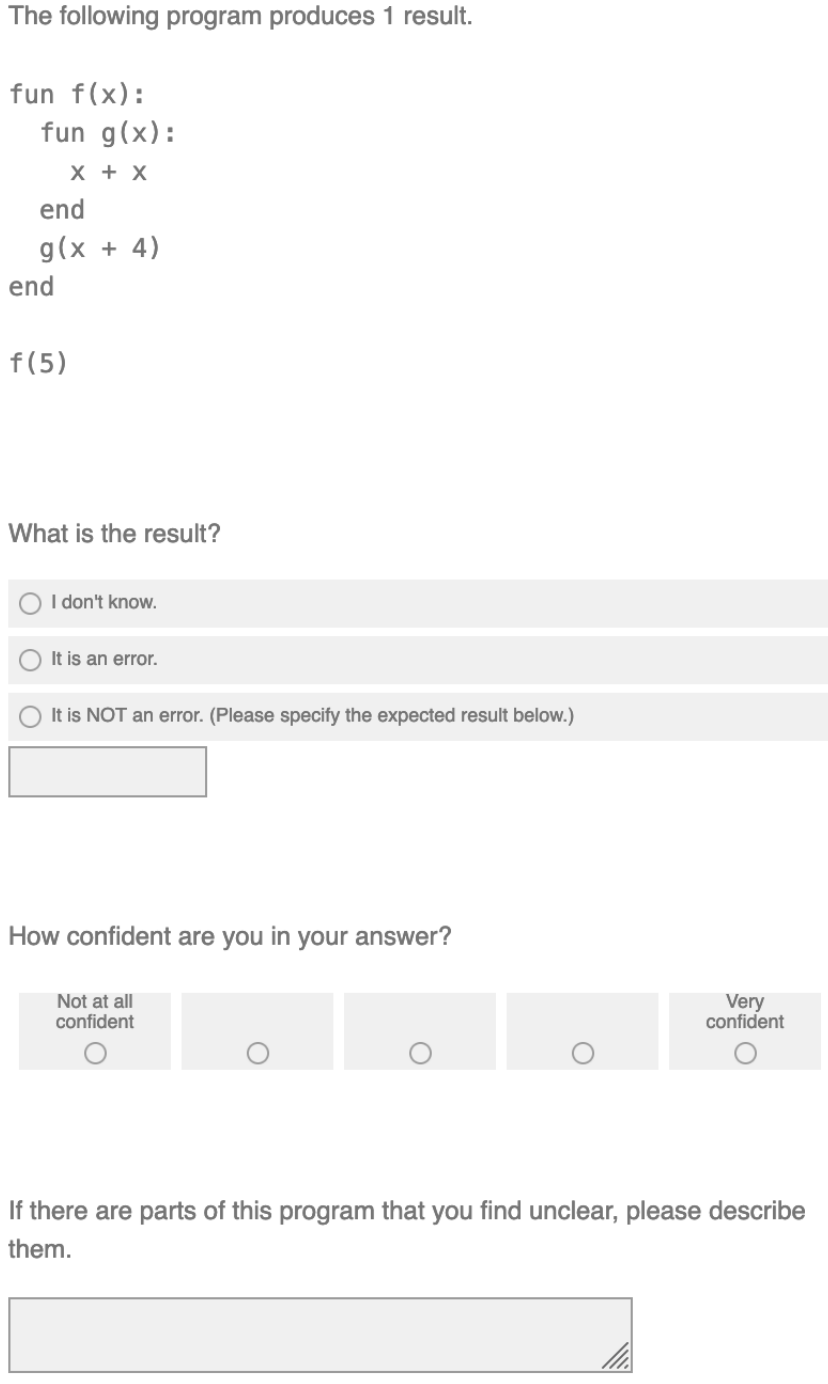

Fig. 7. Racket-Pyret: Survey 1: Per-Pyret Program Output (The first two questions are repeated for every program output.)

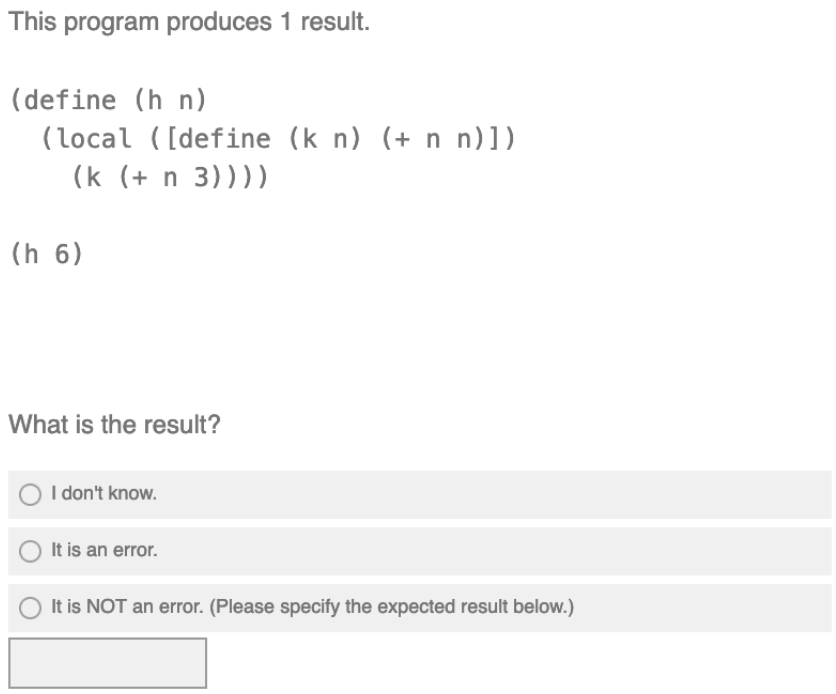

Fig. 8. Racket-Pyret: Survey 1: Per-Racket Program Output (the what-result question is repeated for every program output)

Hypothetical Language:

```
fun f(x, y):
  string-length(x) - string-length(y)
end

f("hello", "world")
f("hi", "there")
```

Racket:

```
(define (h s t)
  (- (string-length s)
     (string-length t)))

(h "minus" "plus")
(h "hey" "hello")
```

In what ways do these programs seem similar and in what ways different?

Do you find one of these programs clearer than the other? If so, why?

You are welcome to comment on any other experiences (programming in other languages, other subjects, etc.) that influence how you understand these programs.

Fig. 9. Racket-Pyret: Survey 1: Program Pairs

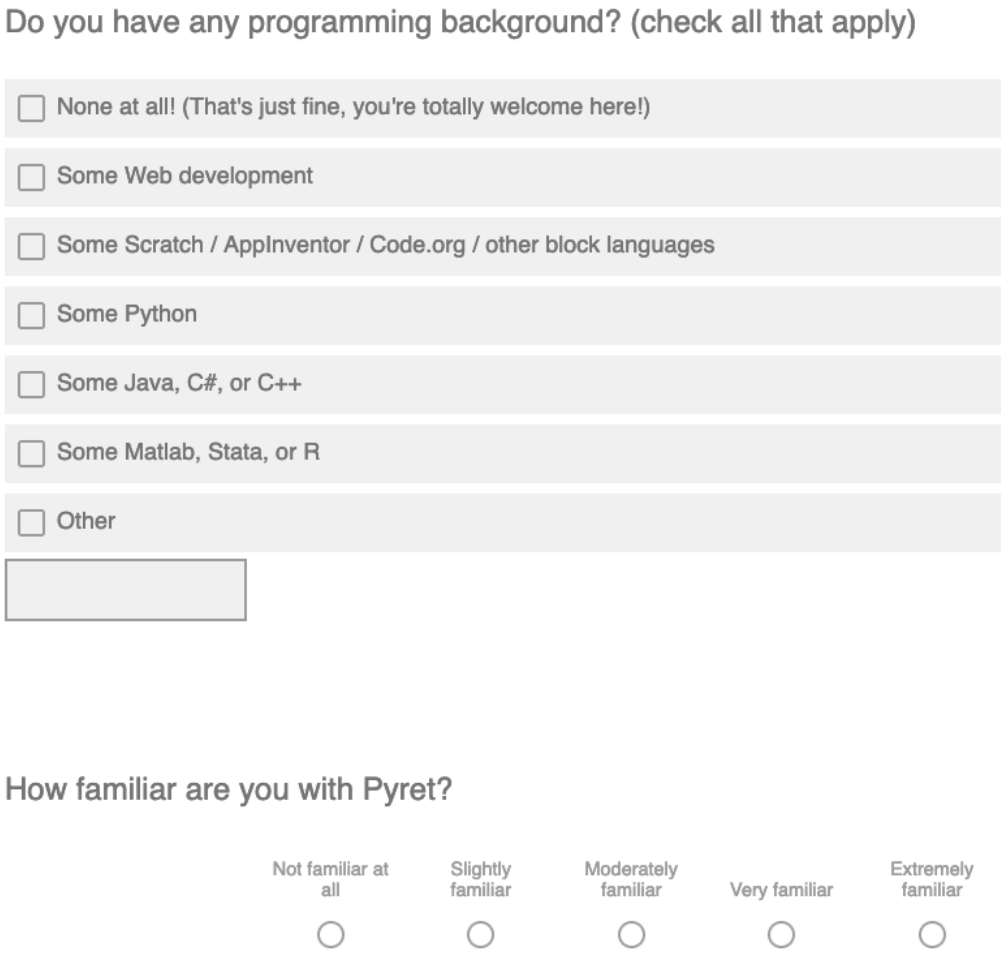

Fig. 10. Racket-Pyret: Survey 1: Background (elides deanonymizing question)

Table 2. Racket-Pyret: Survey 1: Programs Used (no odd linebreaks in original)

ID	Racket	Pyret

1
Fun
	`(define (h s t)`
	`  (- (string-length s)`
	`  (string-length t)))`
	
	`(h "minus" "plus")`
	`(h "hey" "hello")`
		`fun f(x, y):`
		`  string-length(x) - string-`
		`  length(y)`
		`end`
		
		`f("hello", "world")`
		`f("hi", "there")`

2
ImpShadow
	`(define (h n)`
	`  (local ([define (k n) (+ n`
	`  n)])`
	`  (k (+ n 3))))`
	
	`(h 6)`
		`fun f(x):`
		`  fun g(x):`
		`  x + x`
		`  end`
		`  g(x + 4)`
		`end`
		
		`f(5)`

3
ExpShadow
	`(define (h n)`
	`  (local ([define (k n) (+ n`
	`  n)])`
	`  (k (+ n 3))))`
	
	`(h 6)`
		`fun f(x):`
		`  fun g(shadow x):`
		`  x + x`
		`  end`
		`  g(x + 4)`
		`end`
		
		`f(5)`

4
CaseList
	`(define (h m x)`
	`  (if (empty? m)`
	`  0`
	`  (+ (* (first m) x)`
	`  (h (rest m) (* x`
	`  10)))))`
	
	`(h '(4 2 6) 1)`
	`(h '(5 3) 1)`
		`fun g(l, n):`
		`  cases (List) l:`
		`  | empty => 0`
		`  | link(f, r) =>`
		`  (f * n) + g(r, n * 10)`
		`  end`
		`end`
		
		`g([list: 2, 3, 5], 1)`
		`g([list: 7, 8], 1)`

5
SymStr
	`(define (f m)`
	`  (cond`
	`  [(empty? m) false]`
	`  [(cons? m) (or (equal? (`
	`  first m) "stone")`
	`  (f (rest`
	`  m)))`
	`  ]))`
	
	`(f '(x stone z))`
	`(f '(x y z))`
		`fun g(l):`
		`  cases (List) l:`
		`  | empty => false`
		`  | link(f, r) => (f == "`
		`  hello") or g(r)`
		`  end`
		`end`
		
		`g([list: "a", "hello", "c"])`
		`g([list: "a", "b", "c"])`

6 **FirstRest**	`(first (rest (list 9 8 7)))`	`[list: 1, 2, 3].first.rest`
7 **DotFirst**	`(first (cons 6 (cons 5 empty` `  )))`	`link(1, link(2, empty)).` `  first`

8
DefTwice
	`(define z 2)`
	`(define z 1)`
	`z`
		`w = 3`
		`w = 4`
		`w`

9
EarlyUse
	`(define a b)`
	`(define b 5)`
	`a`
		`w = z`
		`z = 3`
		`w`

10
Struct1
	`(define-struct q (x y))`
	
	`(define (h q1 q2)`
	`  (+ (* (q-x q1) (q-y q2))`
	`  (* (q-y q1) (q-x q2))))`
	
	`(h (q 9 8) (q 7 6))`
		`data P: p(a, b) end`
		
		`fun f(p1, p2):`
		`  (p1.a * p2.b) + (p1.b * p2`
		`  .a)`
		`end`
		
		`f(p(2, 3), p(4, 5))`

11
Struct2
	`(define-struct p (x y))`
	`(define-struct t (x y))`
	
	`(define p1 (make-p 9 8))`
	`(define t1 (make-t 1 2))`
	
	`(define (get-ps-x-field a-p)`
	`  (p-x a-p))`
	
	`(get-ps-x-field t1)`
		`data Q: q(a, b) end`
		`data S: s(a, b) end`
		
		`q1 = q(3, 4)`
		`s1 = s(5, 6)`
		
		`fun get-qs-a-field(a-q):`
		`  a-q.a`
		`end`
		
		`get-qs-a-field(s1)`

12
CaseStruct
	`(define-struct b (x y))`
	`(define-struct c (z))`
	
	`(define (g s)`
	`  (cond`
	`  [(b? s) (+ (* (b-x s) (b`
	`  -x s)) (b-y s))]`
	`  [(c? s) (sqrt (c-z s))])`
	
	`(g (make-b 3 0))`
	`(g (make-c 16))`
		`data N:`
		`  | r(a, b)`
		`  | s(c)`
		`end`
		
		`fun h(q):`
		`  cases (N) q:`
		`  | r(e, f) => (e * e) + f`
		`  | s(g)  => num-sqrt(g)`
		`  end`
		`end`
		
		`h(r(2, 3))`
		`h(s(25))`

13
LocalDef
	`(define (f a b)`
	`  (local ([define g (* a a)]`
	`  [define h (+ b b)`
	`  ])`
	`  (- (* g h)`
	`  (+ g h))))`
	
	`(f 4 2)`
		`fun g(x, y):`
		`  q = x * x`
		`  r = y + y`
		`  (q * r) - (q + r)`
		`end`
		
		`g(3, 4)`

14
DotField
	`(define-struct p (x))`
	`(map p-x (list (make-p 6) (`
	`  make-p 5) (make-p 4)))`
		`data S: s(c) end`
		`map(.c, [list: s(1), s(2), s`
		`  (3)])`

15
BasicCond
	`(define (g x)`
	`  (cond`
	`  [(< x 25) "slow"]`
	`  [(= x 25) "normal"]`
	`  [(> x 25) "fast"]))`
	
	`(g 23)`
	`(g 25)`
	`(g 34.5)`
		`fun f(n):`
		`  ask:`
		`  | n < 10 then: "low"`
		`  | n == 10 then: "medium"`
		`  | n > 10 then: "high"`
		`  end`
		`end`
		
		`f(3)`
		`f(10)`
		`f(14.1)`

16
BadCond
	`(define (f x)`
	`  (cond`
	`  [(< x 5)  "C1"]`
	`  [(even? x) "C2"]`
	`  [(> x 5)  "C3"]))`
	
	`(f 16)`
	`(f 5)`
		`fun g(n):`
		`  ask:`
		`  | n < 6 then: "G1"`
		`  | is-odd(n) then: "G2"`
		`  | n > 6 then: "G3"`
		`  end`
		`end`
		
		`g(9)`
		`g(6)`

17
Check
	`(define (h x)`
	`  (+ (* x x) 5))`
	
	`(check-expect (h 2) 9)`
	`(check-expect (h 8) 46)`
		`fun f(n):`
		`  (n * n) + 3`
		`end`
		
		`check:`
		`  f(4) is 19`
		`  f(7) is 46`
		`end`

18
Hyphen
	`(define an 1)`
	`(define s 5)`
	`(define an-s 36)`
	
	`(- an-s an s)`
		`my = 3`
		`x = 4`
		`my-x = 17`
		
		`my-x - my - x`

19 **Caret**	(**define** a 2) (**define** b 6) (**define** a^b (+ 2 6)) a^b	c = 3 d = 4 c^d = 3 + 4 c^d

B RACKET-PYRET: REFUTATION TEXT

Starts on the next page. “EdStem” is the class discussion forum (https://edstem.org/).

Hypothetical Language

As some of you might have guessed, the “hypothetical” language isn’t hypothetical at all. It’s called *Pyret*, and it’s the language we will be using this semester. Every program you saw is pretty much directly runnable in Pyret.

This year, we are trying out a new implementation of Pyret, called *Repartee*, whose language differs slightly from that in previous years, whose implementation we call *CPO*. Below, some of the Pyret code you see has yellow highlights. The survey code (without yellow) runs only in CPO, while the full code (including yellow) will run in both. Just copy the whole program (yellow bits included) into Repartee and you should be fine. (If you’re curious about the innovations in Repartee, see this paper.)

As you may have also guessed from the survey, Pyret is heavily inspired by Racket; indeed, it is almost just “Racket with conventional syntax”. Therefore, instead of explaining Pyret in great detail from scratch, we break up the programs into two groups below:

- One group is programs that behave the same (and for the same reasons) between the two languages. For these we do not offer any commentary, assuming your knowledge from Racket can carry over to Pyret, unless we spotted remarks in your responses that we think need explanation.
- The other is those that behave differently. For these, we provide an explanation of how the two languages differ.

[Both links just point to later in this document. You can also use the outline at the left.]

When in doubt, you can always run both the Racket and Pyret programs! It would be a good idea to run all the Pyret programs to get a feel for the language. Please also consult this guide for converting from Racket to Pyret.

One last comment. Many of your comments talked about “math” when referring to Pyret as opposed to Racket. However, math has many syntaxes! For instance, “three” is sometimes written 3, sometimes III, sometimes ३, etc. Similarly, arithmetic operations are sometimes written “infix”, sometimes “prefix”, sometimes “suffix”, etc. These are all the same “math”, just different notational systems. It’s certainly the case that one is based on current mathematical *convention* (which is exactly why Pyret was created), but neither is more “mathematical” than the other.

If you have questions about any of these programs, please ask on EdStem!

Programs That Are Identical

```
(define (h s t)
  (- (string-length s)
     (string-length t)))

(h "minus" "plus")
(h "hey" "hello")

;; 1
;; -2
```

```
—
```

```
fun f(x :: String, y :: String):
  string-length(x) - string-length(y)
end

f("hello", "world")
f("hi", "there")

# 0
# -3
```

Some thought the second expression might produce an error because the result would be smaller than zero. However, that will not happen. Why? From both languages' perspective, each `string-length` computation produces a number; `-` subtracts one number from the other. Therefore, by the time the minus (-) runs, the two `string-length` expressions have been reduced to numbers, and `-` does not "know" that strings were even involved.

Please run this program in the Racket Stepper! (Unfortunately there isn't one for Pyret, but it would have run in the analogous way.) Watch how the second expression reduces to an answer.

```
(define (h n)
  (local ([define (k n) (+ n n)])
    (k (+ n 3))))

(h 6)

;; 18
```

—

```
fun f(x :: Number):
  fun g(shadow x :: Number):
      x + x
  end
  g(x + 4)
end

f(5)

# 18
```

This program is best understood in contrast to this one. That program errors; this one does not because of the extra `shadow` keyword.

```
(define (h m x)
  (if (empty? m)
      0
      (+ (* (first m) x)
         (h (rest m) (* x 10)))))

(h '(4 2 6) 1)
(h '(5 3) 1)

;; 624
;; 35

_
```

```
fun g(l :: List<Number>, n :: Number):
  cases (List) l:
    | empty => 0
    | link(f, r) =>
      (f * n) + g(r, n * 10)
  end
end

g([list: 2, 3, 5], 1)
g([list: 7, 8], 1)

# 532
# 87
```

In Pyret, we could have written a program similar to the Racket version. However, it is instead idiomatic to use `cases` to take apart ("deconstruct") a list. The `List` in `cases` says "I want to break up a `List`" (as opposed to some other datatype). The `f` and `r` are automatically bound to the corresponding parts of the list: `f` to the first element, `r` to the rest of the list. (We are free to choose any names we like; we use `f` and `r` by convention, but as we will see during the semester, not always.) This type of programming construct is called *pattern-matching*.

```
(first (cons 6 (cons 5 empty)))

;; 6

_

link(1, link(2, empty)).first

# 1
```

These programs behave the same way. `link` is the same as `cons`. The `cons` stands for “construct [a list]”; conventionally, these lists have been called “linked lists”, hence in Pyret we call the constructor `link`.

```
(define z 2)
(define z 1)
z

;; z: this name was defined previously and cannot be re-defined in: z

_

w = 3
w = 4
w

# also error before we even get to the third line
```

In both Racket and Pyret, these produce an error. In both languages, a variable can be associated with only one value. When you try to give a variable two different values *at the same time*, you're basically confusing the language and it's asking you to make up your mind!

If you have prior programming experience, you will have other ideas about what such a program should produce. We will get to this in November.

```
(define a b)
(define b 5)
a

;; error (use before define), we never get to the third line
```

```
_
```

```
w = z
z = 3
w

# error
```

In both languages, these are errors for the same reason: when the first line is evaluated, the variable on the right-hand-side does not yet have a value.

```
(define-struct b (x y))
(define-struct c (z))

(define (g s)
  (cond
    [(b? s) (+ (* (b-x s) (b-x s)) (b-y s))]
    [(c? s) (sqrt (c-z s))]))

(g (make-b 3 0))
(g (make-c 16))

;; 9
;; 4
```

—

```
data N:
  | r(a :: Number, b :: Number)
  | s(c :: Number)
end

fun h(q :: N):
  cases (N) q:
    | r(e, f) => (e * e) + f
    | s(g)    => num-sqrt(g)
  end
end

h(r(2, 3))
h(s(25))

# 7
# 5
```

While almost identical, one difference is that Racket creates two top-level structure definitions (`b` and `c`), whereas Pyret creates one new *type* of thing (`N`) which has two *variants* (`r` and `s`). This difference has interesting consequences in programming languages (e.g., there are some parallels to the difference between Java and JavaScript), but they won't matter here.

```
(define (f a b)
  (local ([define g (* a a)]
          [define h (+ b b)])
      (- (* g h)
      (+ g h))))

(f 4 2)

;; 44
```

—

```
fun g(x :: Number, y :: Number):
  q = x * x
  r = y + y
  (q * r) - (q + r)
end

g(3, 4)

;; 55
```

Racket and Pyret are very similar in this regard: both allow local definitions. Variables and functions defined inside a function body are visible only inside that body; they are invisible outside it (Try accessing one of the locally defined variables after the function call!)

```
(define (g x)
  (cond
    [(< x 25) "slow"]
    [(= x 25) "normal"]
    [(> x 25) "fast"]))

(g 23)
(g 25)
(g 34.5)

;; "slow"
;; "normal"
;; "fast"
```

—

```
fun f(n :: Number):
  ask:
    | n < 10 then: "low"
    | n == 10 then: "medium"
    | n > 10 then: "high"
  end
end

f(3)
f(10)
f(14.1)

# low
# medium
# high
```

`ask` is how you write `cond` in Pyret. Each condition is preceded by a "stick" (`|`) and followed by `then:`.

```
(define (f x)
  (cond
     [(< x 5)   "C1"]
     [(even? x) "C2"]
     [(> x 5)   "C3"]))

(f 16)
(f 5)

;; "C2"
;; error
```

—

```
fun is-odd(n :: Number) -> Boolean:
  num-modulo(n, 2) == 1
end

fun g(n :: Number):
  ask:
    | n < 6 then: "G1"
    | is-odd(n) then: "G2"
    | n > 6 then: "G3"
  end
end

g(9)
g(6)

# "G2"
# error
```

In both languages, conditions are tried in order: the second is tried only if the first fails, etc. (Use the Racket stepper to see how the Racket version evaluates!) Because `is-odd` is not built into Pyret, we have provided its definition.

```
(define an 1)
(define s 5)
(define an-s 36)

(- an-s an s)

;; 30
```

—

```
my = 3
x = 4
my-x = 17

my-x - my - x

# 10
```

In both Racket and Pyret, `my-x` is a single variable name: it is not `x` subtracted from `my`. In Racket that subtraction would be written as `(- my x)`; in Pyret as `my - x` (note the *required* spaces around the subtraction operator). Again, the Racket Stepper can help resolve confusion.

Programs That Differ in Behavior

```
(define (h n)
  (local ([define (k n) (+ n n)])
    (k (+ n 3))))

(h 6)

;; 18
```

—

```
fun f(x):
  fun g(x):
    x + x
  end
  g(x + 4)
end

f(5)

# errors due to shadow binding
```

In principle, these programs are the same. However, in Pyret we chose to add a rule that says if an inner variable has the same name as an outer variable, that is an error. This catches a pattern where programmers write such names by accident, but when using it fail to realize which of the two variables they're getting, *think* it's the outer one when it's actually the inner one, and not only end up with buggy programs but also get very confused when trying to find the error.

Often, the simple fix is to just use a different name. Sometimes, you will find that you *want* to hide the outer name so as to not access it accidentally. Pyret will let you do that, so long as you make clear you want that feature. This feature is called "shadowing", so Pyret asks you to use the `shadow` keyword, as in that program.

```
(define (f m)
  (cond
     [(empty? m) false]
     [(cons? m) (or (equal? (first m) "stone")
                    (f (rest m)))]))

(f '(x stone z))
(f '(x y z))

;; false (because symbol vs string)
;; false
```

_

```
fun g(l :: List<String>):
  cases (List) l:
     | empty => false
     | link(f, r) => (f == "hello") or g(r)
  end
end

g([list: "a", "hello", "c"])
g([list: "a", "b", "c"])

# true
# false
```

These programs look very similar, but are subtly different. In Racket, the content of the lists are *symbols*; in Pyret, they're *strings*. Because in Racket strings are different from symbols, comparing `'stone` with `"stone"` will give `false`. In Pyret, `"hello"` compared to `"hello"` gives `true`. By design, Pyret does not have symbols.

```
(first (rest (list 9 8 7)))

;; 8
```

```
_

[list: 1, 2, 3].first.rest

# error
```

In both languages, functions apply from the “inside out”: we could, for instance, have rewritten the Pyret program as

```
([list: 1, 2, 3].first).rest
```

to make this more explicit. The first element of this list is 1. That is a number, not a list, so it does not have a `rest`. Therefore, this program results in an error. The version that is equivalent to the Racket program would be

```
[list: 1, 2, 3].rest.first
```

or

```
([list: 1, 2, 3].rest).first
```

Don’t get thrown off by the fact that the names seem to be in the “opposite order”.

(The difference between Racket and Pyret is similar to the difference between “the first element **of** the list” and “the list**’s** first element”. Thus, the two correct versions correspond to “the first element of the rest of the list” [Racket] and “the list’s rest’s first element” [Pyret].)

```
(define-struct q (x y))

(define (h q1 q2)
  (+ (* (q-x q1) (q-y q2))
     (* (q-y q1) (q-x q2))))

(h (q 9 8) (q 7 6))

;; ERROR! constructor in *SL requires make-p

_

data P: p(a :: Number, b :: Number) end

fun f(p1 :: P, p2 :: P):
  (p1.a * p2.b) + (p1.b * p2.a)
end

f(p(2, 3), p(4, 5))

;; also 22
```

When making new data, the student languages of Racket require you to write the `make-` prefix. Pyret does not require this. (Neither does the full Racket language!) If you've gotten used to writing `make-`, you may run into errors in Pyret.

```
(define-struct p (x y))
(define-struct t (x y))

(define p1 (make-p 9 8))
(define t1 (make-t 1 2))

(define (get-ps-x-field a-p)
  (p-x a-p))

(get-ps-x-field t1)

;; ERROR: p-x: expects a p, given (make-t 1 2)
```

—

```
data Q: q(a :: Number, b :: Number) end
data S: s(a :: Number, b :: Number) end

q1 = q(3, 4)
s1 = s(5, 6)

fun get-qs-a-field(a-q :: {a :: Number}):
  a-q.a
end

get-qs-a-field(s1)

# 5
```

This pair of programs point to an important difference in programming language design. Observe that each pair of data definitions has the same number of fields with the same names and contracts. Are they therefore interchangeable?

The contract `{a :: Number}` means *any* value that has a field name a with type `Number` can be passed to `get-qs-a-field`. Because the contract expects only that, not a `Q` (despite what the function and variable names imply), we can pass an instance of `S`. Had the contract instead read `Q`, then the program would error. This error is analogous to that in Racket, but it happens *before* the program starts to run, and hence the error report is in a different place and style.

```
(define-struct p (x))

(map p-x (list (make-p 6) (make-p 5) (make-p 4)))

;; (list 6 5 4)

_

data S: s(c :: Number) end

map(.c, [list: s(1), s(2), s(3)])

# syntax error
```

Conceptually, these programs are very similar. However, writing just `.c` is a *syntax* error in Pyret: a `.`-accessor must always be preceded by some expression. One can instead write (the new parts are in green):

```
map(lam(x): x.c end, [list: s(1), s(2), s(3)])
```

(which is just like the `lambda` in Racket) or equivalently but more concisely:

```
map({(x): x.c}, [list: s(1), s(2), s(3)])
```

or even more concisely:

```
map(_.c, [list: s(1), s(2), s(3)])
```

The _ automatically turns this program into the equivalent ones above: creating a one-argument anonymous function and using its parameter in place of _. While _ is very elegant in situations like this, it can get confusing in more complex contexts, so use it with caution, and feel free to use explicit functions in its place.

```
(define a 2)
(define b 6)
(define a^b (+ 2 6))
a^b

;; 8
```

—

```
c = 3
d = 4
c^d = 3 + 4
c^d
```

Though Pyret is quite permissive with its variable names, it isn't as permissive as Racket. You can't, for instance, put ^ in the middle of a name. (In fact, ^ in Pyret is a special operator called the "cannonball". But we won't have much need for it!)

If you're *really* curious: the cannonball operator takes the value from the left of the cannonball and feeds it as the first (and only) argument to the function on the right. For instance,

```
fun sq(n :: Number): n * n end
check:
  3 ^ sq is 9
end
```

What if you have a multi-arity function? As long as you can turn it into a single-arity function, you can still cannonball. The _ construct can be especially useful for this:

```
check:
  3 ^ num-expt(_, 2) is 9
end
```

In practice, we rarely write a constant to the left of ^, but rather an expression that computes a result. Thus, cannonballing lets you chain sequences of functions: e.g.,

```
check:
  sq(4) ^ num-expt(_, 2) ^ num-sqrt is 16
end
```

```
;; f :: Number -> List-of-Number
(define (f x)
  (list x x))

(f "okay")

;; (list "okay" "okay")
```

—

```
fun g(y :: Number) -> List<Number>:
  y + 3
end

g("hello")

# ERROR: number annotation not satisfied
```

In the student languages of Racket (as opposed to full Racket), contracts are written as comments, so they are not checked. In Pyret, they are part of the program's syntax, and are explicitly checked by the language. Therefore, the Pyret program produces an error because `"hello"` is not a `Number`.

```
;; f :: List-of-Number -> List-of-List-of-Number
(define (f x)
  (list x x))

(f (list "okay"))

;; (list (list "okay") (list "okay"))
```

—

```
fun g(l :: List<Number>) -> List<List<Number>>:
  [list: l, l]
end

g([list: "hello"])

# [list: [list: "hello"], [list: "hello"]]
```

This program is a bit unusual because the two versions of Pyret will behave differently. Either way, Racket will not produce an error because contracts are comments, and hence not checked. In the CPO version of Pyret, the checks go only “one level down”, so this program does *not* signal an error (the list of strings matches the criterion of being a list). In the Repartee version of Pyret, checks go “all the way down”, so this program *does* signal an error.

```
(define (h x)
  (+ (* x x) 5))

(check-expect (h 2) 9)
(check-expect (h 8) 46)
```

—

```
fun f(n :: Number):
  (n * n) + 3
end

check:
  f(4) is 19
  f(7) is 46
end
```

These programs are essentially the same. However, syntactically, Pyret allows multiple tests to be written in a single `check` block (`examples` can equivalently be written in place of `check`). In addition, Pyret lets you "name" a block with a descriptive string, as below:

```
check "addition":
  1 + 1 is 2
end

check "subtraction":
  3 - 3 is 0
  4 - 3 is 1
end
```

In CPO, each block's results are reported separately and using the name. (Try it out!) Repartee does not *yet* support this (this may change over the course of the semester).

Splitting tests into blocks does not change the program's behavior. However, as the number of tests grows larger, having names for test blocks can be very helpful at localizing faults. This also greatly helps a person reading your program. Therefore, we ask you to break your tests into meaningful blocks and give each block a name that reflects its intent.

C RACKET-PYRET: SURVEY 2

To summarize, Survey 2 presented material in the following order:

(1) Survey explanation: Figure 11.
(2) Questions: Figure 12.

All the programs are in Table 3.

To prepare for the semester, we are going to show you a handful of Pyret programs and ask you to tell us what they produce.

We ask you to answer these *without running* the programs. It's okay if you're wrong! Any incorrectness in your responses will **NOT** affect your grade or anything else. We only ask you to treat these questions seriously. They are an important part of your learning process, and will help us determine which parts of Pyret we need to further explain (so cheating on this will only hurt you!).

You can have the same answer for multiple questions.

We estimate that the entire survey will take you about 10 minutes. You are welcome to stop at any time and resume later; the system will keep track of where you were.

Please complete the survey by 11:59 PM, **Wednesday, September 7**.

Fig. 11. Racket-Pyret: Survey 2: Opening Page

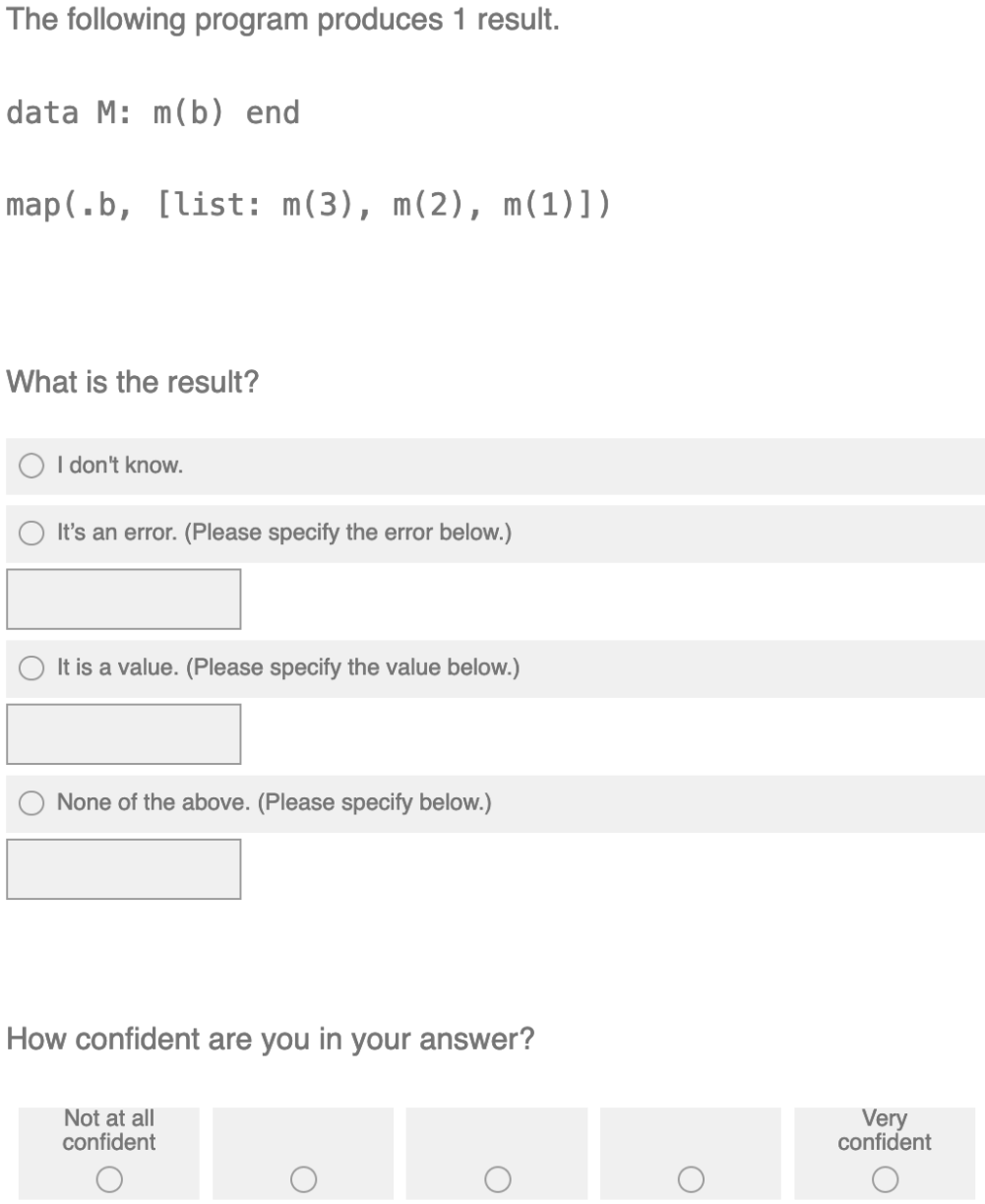

Fig. 12. Racket-Pyret: Survey 2: Per-Program

Table 3. Racket-Pyret: Survey 2: Programs Used

ID	Pyret

1	`fun p(n):`
	`  fun h(shadow n):`
	`  n + n`
	`  end`
	`  h(n + 2)`
	`end`
	
	`p(300)`

2	`u = 8`
	`w = 3 + u`
	`u = 4`
	`x = 5`
	`u`

3	`data M: m(b) end` `map(.b, [list: m(3), m(2), m(1)])`

D RACKET-PYRET: DATA ANALYSIS

We identified the most "challenging" Pyret programs by looking at their distributions over various measurements. Roughly speaking, we considered a program challenging if many students failed to predict the correct output or many students found it confusing.

D.1 Identifying Challenging Programs

Some of the programs have multiple top-level expressions, leading to multiple outputs. We refer to each output separately as a *task*. In total, there are 25 tasks in Racket and 26 in Pyret (because one one-output Racket program is paired with two Pyret programs).

We computed the following data:

CorrectCount how many students correctly[2] predicted a task's output
IDKCount how many students responded "I don't know" (IDK) for a task
AverageConfidence for a task, how confident students were in their predictions on average
UnclearCount how many students responded to the anything-unclear question

Figure 13 labels the programs we considered the most challenging. Numbers in the first three sub-plots can be found in Table 4. In the first three sub-figures the choice is probably clear. For UnclearCount, we noticed that students seemed more likely to write descriptive text about their confusion for earlier programs than later ones. We conjectured that this may be because students got tired near the end of a long list of programs. Applying Kendall's test [Kendall 1938] confirmed (p-value < 0.05) that the presence of text statistically correlates with the position for *all* the textual answers (earlier questions were more likely to have text) with one exception, where the question "In what ways do these programs seem similar/different" was required to be answered by every student for every program. Notably, the other measurements do *not* correlate with position. Thus, we picked the high UnclearCount's *relative to position*.

D.2 Racket-Pyret: The Effect of Prior Knowledge

We analyzed students' Pyret correctness against their Racket correctness and their self-declared prior programming background. Specifically, for each student and each Racket-Pyret program pair, we considered the following variables:

IsPyretCorrect whether the student predicted the Pyret program correctly
ProgramID which program the question is about
IsRacketCorrect whether the student predicted the Racket program correctly
HasXBackground several boolean variables that encode whether the student had background in Web/Block-based languages/Python/etc.

Each of the other variables might or might not contribute to the prediction of the outcome, **IsPyretCorrect**. So in principle, there is an exponential number of models/theories to compare. We only compare the following models/theories, which seem to be most relevant in this context:

(1) Students have a uniform probability of giving correct answers regardless of all other variables.
(2) Students have different correctness rates for different **ProgramID**s, but none of the other variables has an influence.
(3) Students have different correctness rates depending only on their Racket correctness. That is, students who predicted a Racket program correctly are more likely to predict the corresponding Pyret program correctly.

[2]A prediction doesn't have to be an exact match to be considered correct. We cleanse the data for typos, case, varieties of writing Booleans and lists, etc.

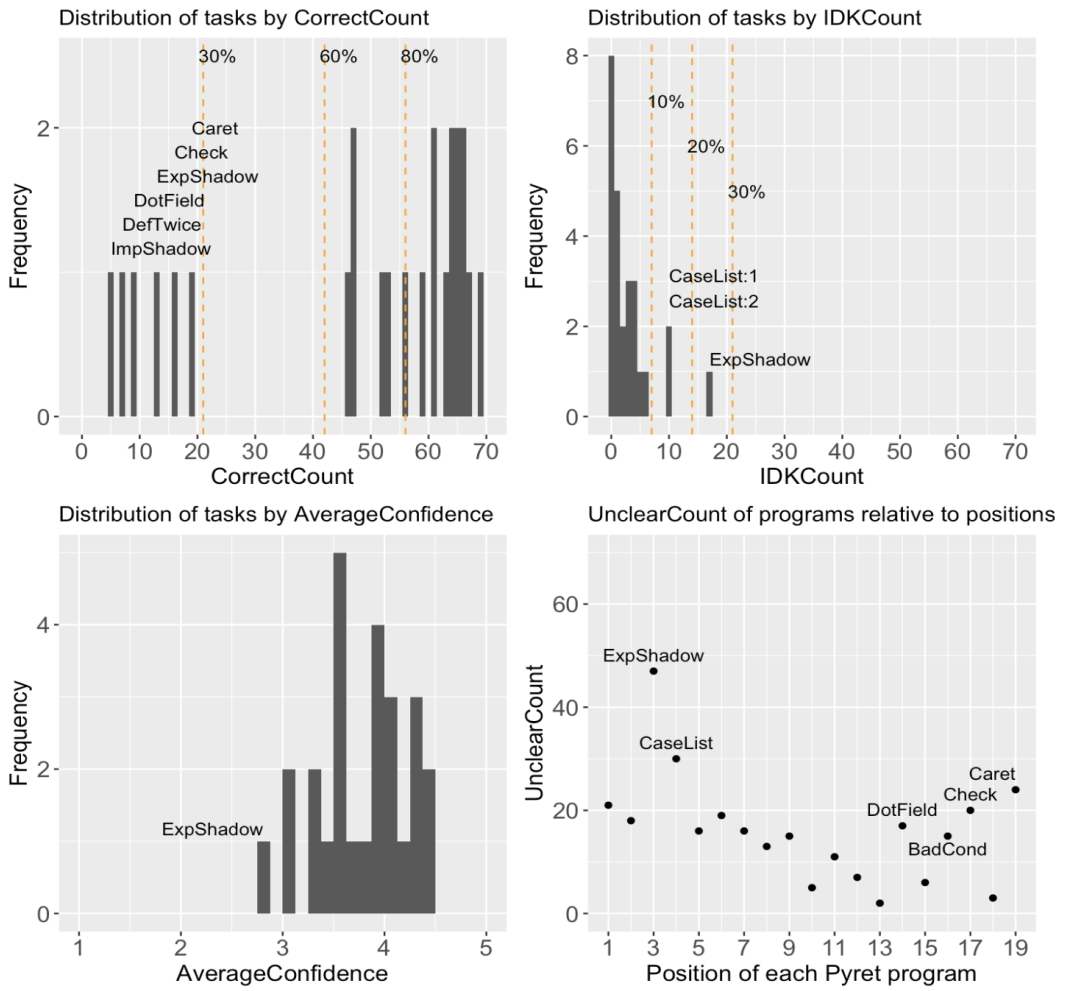

Fig. 13. Racket-Pyret: Survey 1: The Most Challenging Pyret Programs

(4) Students have different correctness rates depending only on one background variable. That is, students with that kind of background generally do better (or worse). There are several models under this category, one for each kind of background.
(5) The correctness rate depends on both the program and the student's Racket correctness. The two variables might or might not interact: the impact of Racket correctness might or might not be uniform on all programs. Thus, there are two models are under this category.
(6) The correctness rate depends on both the program and whether the student has one kind of background. Again, the variables might or might not interact.

We compare these models by their Akaike Information Criterion (AIC), which is a non-negative real number that measures how well a model fits a data. AIC penalizes models with more parameters. Therefore, it strives for a balance between goodness of fit and the model's complexity. Models with smaller AIC are better.

Table 4. Racket-Pyret: Survey 1: Pyret Results

Task	CorrectCount	IDKCount	AverageConfidence
Fun:1	66 (94%)	0	4.24
Fun:2	63 (90%)	0	4.09
ImpShadow	5 (7%)	1	4.04
ExpShadow	13 (19%)	17	2.81
CaseList:1	47 (67%)	10	3.58
CaseList:2	46 (66%)	10	3.57
SymStr:1	61 (87%)	4	3.91
SymStr:2	61 (87%)	4	3.89
FirstRest	52 (74%)	5	3.05
DotFirst	67 (96%)	3	3.52
DefTwice	7 (10%)	1	3.93
EarlyUse	53 (76%)	2	3.62
Struct1	66 (94%)	0	4.29
Struct2	59 (84%)	3	3.52
CaseStruct:1	65 (93%)	1	3.83
CaseStruct:2	65 (93%)	1	3.88
LocalDef	64 (91%)	0	4.34
DotField	9 (13%)	6	3.34
BasicCond:1	70 (100%)	0	4.49
BasicCond:2	70 (100%)	0	4.50
BasicCond:3	69 (99%)	0	4.34
BadCond:1	56 (80%)	1	3.68
BadCond:2	47 (67%)	2	3.50
Check	16 (23%)	3	3.37
Hyphen	64 (91%)	0	4.07
Caret	19 (27%)	4	3.11

Table 5. Racket-Pyret: Survey 1: Model Comparison

Variables	**AIC**
(Nothing)	2215.3
ProgramID	1362.8
IsRacketCorrect	2109.9
HasXBackground	[2216.9, 2217.3]
ProgramID + IsRacketCorrect	1314.2
ProgramID ∗ IsRacketCorrect	1296.4
ProgramID + HasXBackground	[1364.1, 1364.8]
ProgramID ∗ HasXBackground	[1376.3, 1389.7]

It is subtle to compare two models that have close AIC. Burnham and Anderson [2004] suggest the following rule of thumb: "models having $\Delta_i \leq 2$ have substantial support (evidence), those in which $4 \leq \Delta_i \leq 7$ have considerably less support, and models having $\Delta_i > 10$ have essentially no support." Here Δ_i refers to the AIC difference between a model and the best model under consideration.

The AICs (Table 5) suggest that both Racket and the programs themselves impact students' correctness in Pyret, that the programs have bigger impact over Racket, and that the impacts are non-orthogonal. Overall, taking programming background into account unnecessarily complicates the model.

Interestingly, the ProgramID $*$ IsRacketCorrect model achieves a better (smaller) AIC than the similar model without interaction. This suggests that the transfer from Racket to Pyret is uneven in the studied programs. A closer look at the estimated parameters of the with-interaction model suggests that, although the transfer is positive in general (p-value = 0.01), the transfer is clearly harmful in **DotField** (p-value = 0.02) and in **Caret** (p-value $<$ 0.001).

We further investigated the impact of programming background on the **DefTwice** program, which can be deeply influenced by its syntactic similarity to Python. Curiously, Python (AIC = 49.43643) does not beat the empty model (AIC = 47.51162), but one other programming background stands out: block languages (AIC = 44.13359). However, the impact of this background is still *not* statistically significant (p-value = 0.05336).

The statistical analysis was done with the `glm` function from R [R Core Team 2022]. The `family` parameter is set to `binomial` (i.e., the link function is logistic) because the outcome is binary.

E PYRET-PYTHON: PHASES 1 & 2

Phases 1 & 2 have similar structure. Phase 1 starts with an instruction (Figure 14), followed by questions about program outputs (illustrated with Figure 16), and finally a question about Python background (Figure 17). Phase 2 has essentially the same instruction (Figure 15), the exact same format for the major questions, and does not repeat the question about Python background. All programs are listed in Table 6.

Q1 Program 1

1 Point

Instructions

To prepare for the transition from Pyret to Python, we are going to show you several Python programs and ask you to tell us what you EXPECT they will produce.

Please **DO NOT RUN THE PROGRAMS**. The whole point of this is for us to identify what INTUITIONS people hold about these programs as we start the transition. *This is especially valuable from those with no prior Python experience*. If you run the programs to see the "correct" answers, you'll actually interfere with our learning goals for doing this before we start Python.

Question Format

For each question, we are going to show you a Python program entered into the interactions window (which is similar to Pyret's, except the prompt is `>>>` and other lines before the next prompt are marked with `...` (the dots do NOT mean "something to fill in" as in Pyret)).

We've marked zero or more lines in the interaction as `??????`. The questions ask you to tell us what would appear in place of `??????` had we run the program in the interactions window. Select the result you expect; if you really can't decide, you can select multiple answers. If you selected an error or multiple answers, we'll ask you to briefly explain. Otherwise, you can leave those open-response explanations blank.

Fig. 14. Pyret-Python: Phase 1: Instruction

Q1 Program 1

1 Point

Instructions

To help you transition from Pyret to Python, we are going to show you several Python programs and ask you to tell us what you EXPECT they will produce.

Some programs are similar to the programs in the first Python assignment. Other programs use Python constructs that you recently learned.

Please **DO NOT RUN THE PROGRAMS**. The whole point of this is for us to identify what parts of Python are more difficult to get used to. *This is especially valuable from those with less Python experience*. If you run the programs to see the "correct" answers, you'll actually interfere with our learning goals for doing this.

Question Format

For each question, we are going to show you a Python program entered into the interactions window. We've marked zero or more lines in the interaction as `??????`. The questions ask you to tell us what would appear in place of `??????` had we run the program in the interactions window. Select the result you expect; if you really can't decide, you can select multiple answers. If you selected an error or multiple answers, we'll ask you to briefly explain. Otherwise, you can leave those open-response explanations blank.

Fig. 15. Pyret-Python: Phase 2: Instruction

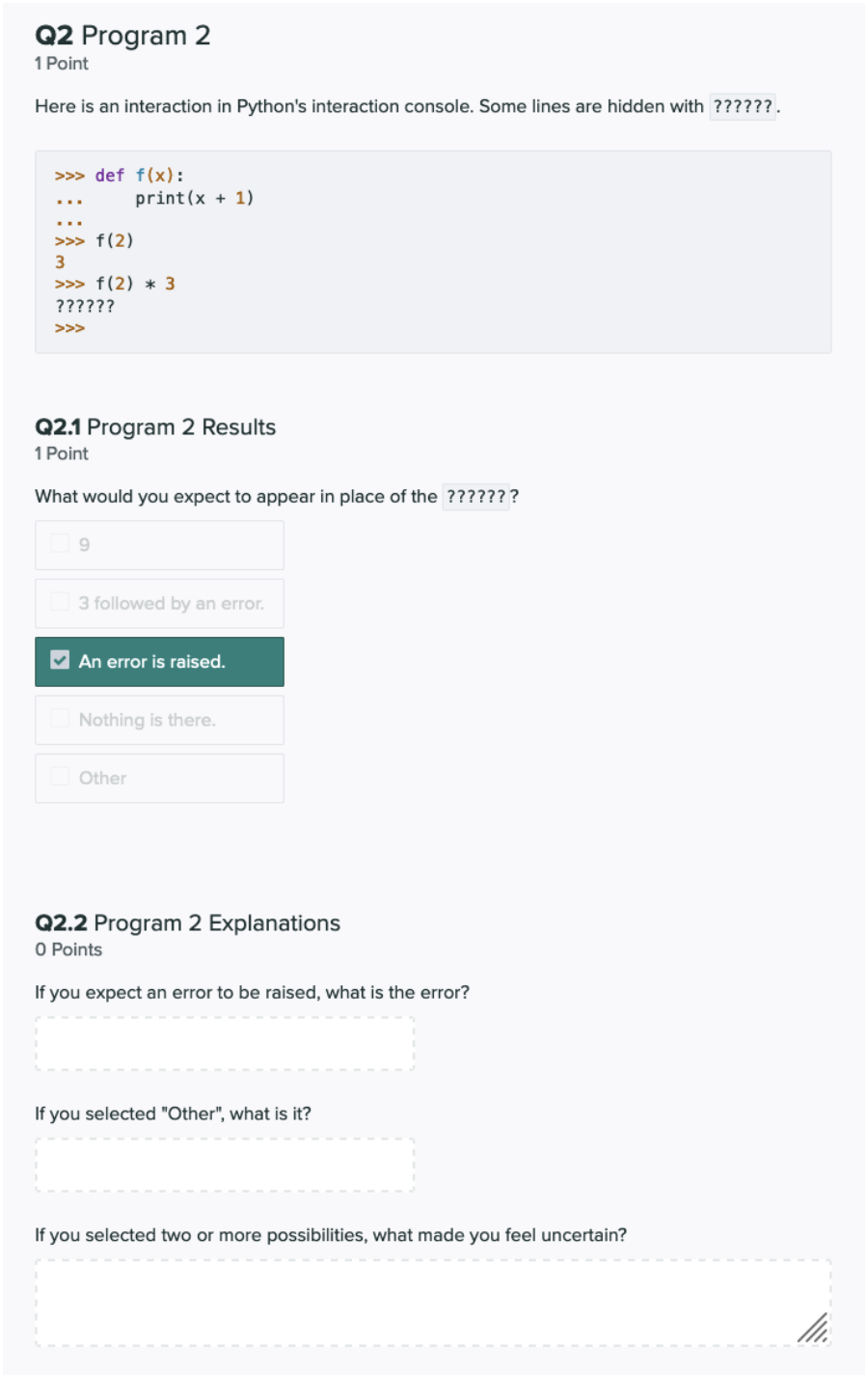

Fig. 16. Pyret-Python: Phases 1 & 2: Major Questions

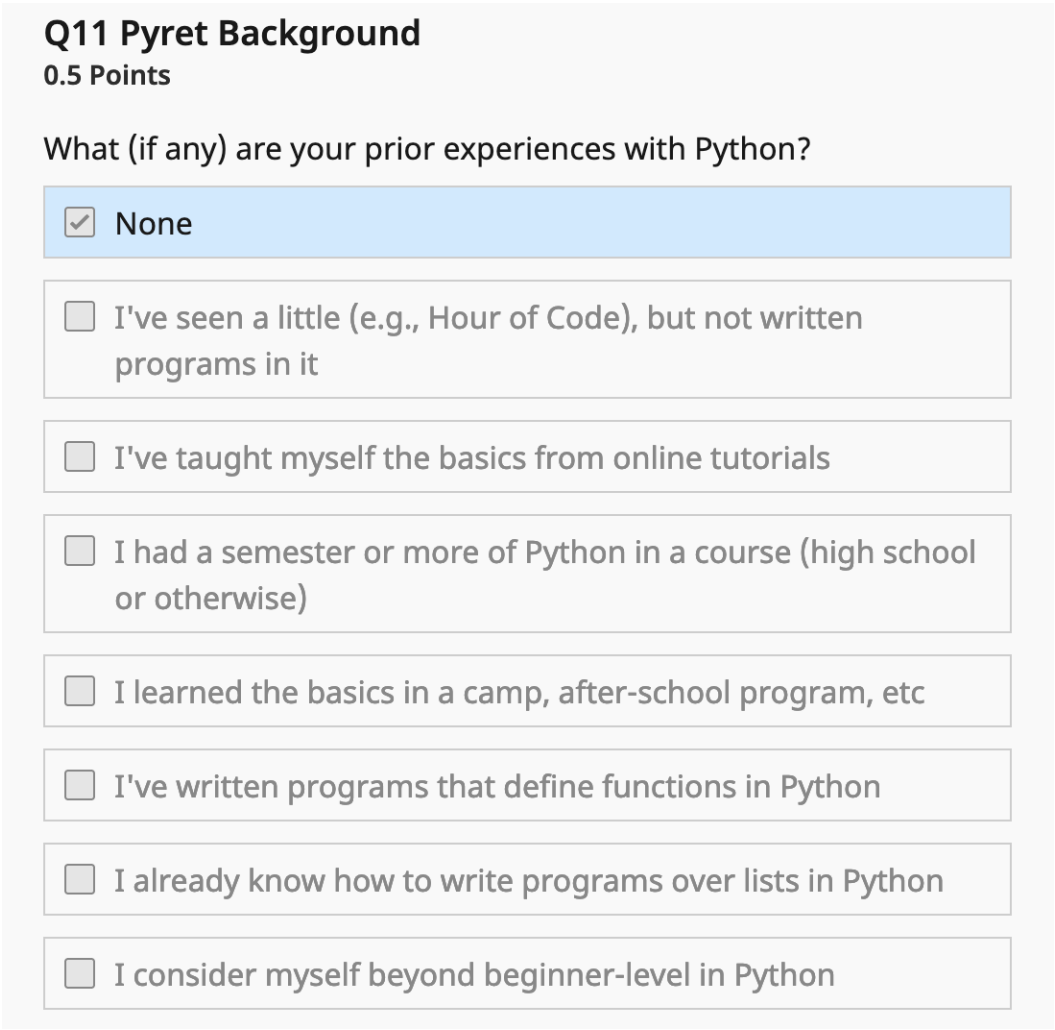

Fig. 17. Pyret-Python: Phase 1: *Python* Background (There is a typo in the question title, which says "Pyret Background". But the question prompt and the choices make clear that we are asking for Python background.)

Table 6. Pyret-Python: Phases 1 & 2: Programs Used

Phase	ID	Python

Phase 1	1	`>>> def f(x):`
		`...  x + 1`
		`...`
		`>>> f(2)`
		`??????`
		`>>>`

Phase 1	2	`>>> def f(x):`
		`...  print(x + 1)`
		`...`
		`>>> f(2)`
		`3`
		`>>> f(2) * 3`
		`??????`
		`>>>`

Phase 1	3	`>>> def f(x):`
		`...  return x + 1`
		`...`
		`>>> f(2)`
		`3`
		`>>> f(2) * 3`
		`??????`
		`>>>`

Phase 1	4	`>>> def f(x):`
		`...  return`
		`...  x + 1`
		`...`
		`>>> f(2)`
		`??????`
		`>>>`

Phase 1	5	`>>> def f(x, y):`
		`...  s = x + y`
		`...  return s / 2`
		`...`
		`>>> f(4, 6)`
		`5`
		`>>> def g(a, b):`
		`...  n = a + b`
		`... return n / 2`
		`??????`
		`>>>`

Phase 1	6	`>>> def f(x: int, y: int):`
		`...  return x + y`
		`...`
		`>>> f(2, 3)`
		`5`
		`>>> f("hel", "lo")`
		`??????`
		`>>>`

Phase 1	7	`>>> def f(x: int, y: int):`
		`...  return x * y`
		`...`
		`>>> f(2, 3)`
		`6`
		`>>> f("hello", 3)`
		`??????`
		`>>>`

Phase 1	8	
		`>>> def f(x):`
		`...  if x < 60:`
		`...  return "too-low"`
		`...  else:`
		`...  return "okay"`
		`...`
		`>>> f(59)`
		`'too-low'`
		`>>> f(60)`
		`'okay'`
		`>>> def g(n):`
		`...  if n < 60:`
		`...  "too-low"`
		`...  else:`
		`...  "okay"`
		`...  return`
		`...`
		`>>> g(59)`
		`??????`

Phase 1	9	
		`>>> def f(x):`
		`...  if x <= 2:`
		`...  y = 10`
		`...  else:`
		`...  y = 100`
		`...  return x * y`
		`...`
		`>>> f(2)`
		`??????`
		`>>>`

Phase 1	10	`>>> x = 2`
		`>>> def get_x():`
		`...  return x`
		`...`
		`>>> get_x()`
		`2`
		`>>> x = 3`
		`??????`
		`>>> get_x()`
		`??????`
		`>>>`

Phase 2	1	`>>> def g(n):`
		`...  2 * n`
		`...`
		`>>> g(3)`
		`??????`
		`>>>`

Phase 2	2	`>>> def g(n):`
		`...  print(2 * n)`
		`...`
		`>>> g(2)`
		`4`
		`>>> g(3)`
		`6`
		`>>> g(3) + 1`
		`??????`
		`>>>`

Phase 2	3	`>>> def g(n):`
		`...  return`
		`...  2 * n`
		`...`
		`>>> g(3)`
		`??????`
		`>>>`

Phase 2	4	
		`>>> def g(n, m):`
		`...  d = n - m`
		`...  return d * d`
		`...`
		`>>> g(3, 5)`
		`4`
		`>>> def f(x, y):`
		`...  t = x - y`
		`... return t * t`
		`??????`
		`>>>`

Phase 2	5	
		`>>> def g(s: str, t: str):`
		`...  return s + t`
		`...`
		`>>> g("py", "thon")`
		`'python'`
		`>>> g(1, 2)`
		`??????`
		`>>>`

Phase 2	6	
		`>>> def g(s: str, n: int):`
		`...  return s * n`
		`...`
		`>>> g("ha", 3)`
		`'hahaha'`
		`>>> g(2, 3)`
		`??????`
		`>>>`

Phase 2	7	
		`>>> def g(n):`
		`...  if n < 18:`
		`...  return "too young"`
		`...  else:`
		`...  return "good"`
		`...`
		`>>> g(17)`
		`'too young'`
		`>>> g(18)`
		`'good'`
		`>>> def h(x):`
		`...  if x < 18:`
		`...  "too young"`
		`...  else:`
		`...  "good"`
		`...  return`
		`...`
		`>>> h(17)`
		`??????`
		`>>>`

Phase 2	8	
		`>>> n = 3`
		`>>> def get_n():`
		`...  return n`
		`...`
		`>>> get_n()`
		`3`
		`>>> n = 2`
		`??????`
		`>>> get_n()`
		`??????`
		`>>>`

Phase 2	9	
		`>>> def f(ns):`
		`...  total = 0`
		`...  for n in ns:`
		`...  total = total + n`
		`...  return total`
		`...`
		`>>> f([2, 3])`
		`??????`
		`>>>`

Phase 2	10	
		`>>> def f(ns):`
		`...  total = 0`
		`...  for n in ns:`
		`...  total = total + n`
		`...  return total`
		`...`
		`>>> f([2, 3])`
		`??????`
		`>>>`

Phase 2	11	
		`>>> def f(ns):`
		`...  total = 0`
		`...  for n in ns:`
		`...  total = total + n`
		`...  print(total)`
		`...`
		`>>> f([2, 3])`
		`??????`
		`>>>`

Phase 2	12	

```
>>> def append_twice(ls, x):
...     ls.append(x)
...     ls.append(x)
...     return ls
...
>>> list_1 = [1, 2]
>>> append_twice(list_1, 3)
[1, 2, 3, 3]
>>> append_twice(list_1, 30)
??????
>>>
```

F PYRET-PYTHON: PHASE 3: CODEBOOK

This appendix presents the codebook that we used to analyze data collected in Phase 3.

CorrectExplanation The response provides a correct explanation. Only an explanation that is compatible with the Python semantics counts as correct.

WrongExplanationCorrectRepair The response provides an incorrect explanation, but nevertheless suggests a correct repair.

WrongExplanation The response provides an incorrect explanation and doesn't suggest a correct repair.

MissingExplanationCorrectRepair The response provides no explanation for a correct repair.

MissingExplanation The response provides no explanation and doesn't suggest a correct repair.

G PYRET-PYTHON: STATISTICAL ANALYSIS

The analysis procedure is the same as in Appendix D.2.

For each student at each predict-the-output question, we consider three variables:

IsCorrect whether the student gave a correct answer
ProgramID which program the question is about
HasBackground whether the student had any Python background

The outcome, **IsCorrect**, might or might not depend on one of the other two factors. This suggests the following models/theories to consider:

(1) Students have a uniform probability of giving correct answers regardless of their backgrounds and regardless of the program.
(2) Students have different correctness rates for different Programs, but Python background has no influence.
(3) Students have different correctness rates depending only on their Python background. That is, students with Python background generally do better (or worse).
(4) The correctness rate depends on both variables, additively. That is, students with Python background are *not* particularly good/bad at some programs.
(5) The correctness rate depends on both variables, with interaction. That is, students with Python background *are* particularly good/bad at some programs.

The results of Phase 1, Phase 2, and the final exam are presented in Tables 7 to 9 respectively. They suggest that prior Python background *does* have an influence on students' correctness rate. The influence might or might not vary in different programs. Furthermore, **HasBackground** has a clear (i.e., statistical significant) positive impact (p-value < 0.001 in Phases 1 & 2; p-value < 0.01 in the final exam) in both additive models. (In the with-interaction models the impact is shared by the interaction parameters and hence becomes unclear.)

Some data from the final exam are excluded for the computation of AIC because those students didn't participate in Phase 1, when we collected their Python background information. We ended up with 160 students.

We don't analyze the results in Phase 3 with similar method because we think the existing results already show a consistent and clear pattern and because the analysis would require us to grade hundreds of text responses.

Table 7. Pyret-Python: Phase 1: Model Comparison

Variables	AIC
(Nothing)	2463.4
ProgramID	1770.9
HasBackground	2453.5
ProgramID + HasBackground	1754.2
ProgramID ∗ HasBackground	1753.9

Received 2023-03-01; accepted 2023-06-27

Table 8. Pyret-Python: Phase 2: Model Comparison

Variables	**AIC**
(Nothing)	2663.2
ProgramID	2214.1
HasBackground	2313.2
ProgramID + HasBackground	1897.8
ProgramID ∗ HasBackground	1910.6

Table 9. Pyret-Python: Final Exam: Model Comparison

Variables	**AIC**
(Nothing)	627.2
ProgramID	479.8
HasBackground	622.5
ProgramID + HasBackground	472.1
ProgramID ∗ HasBackground	475.5