Tutorial 6

11/8/2023

100/100 Points

Attempt 1 | Review Feedback | Offline Score: **100/100** | View feedback

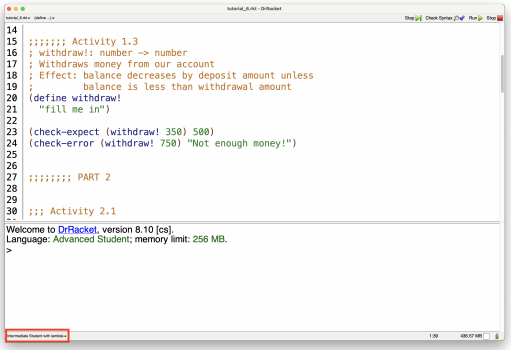
Unlimited Attempts Allowed

11/8/2023

Details

> Note: For groups that may have a significant amount of programming experience, your PM may suggest trying out the **Advanced Tutorial 6 (https://bain-cs111.github.io/assignments/adv-tutorial-6)**

Congratulations, you've graduated to another version of Racket! Select Choose Language from the Language menu (shown below with the **wrong** language selected) and select `Advanced Student`. Now you will be able to do all of the imperative operations we've been talking about this week.



Learning Objectives: For each of the imperative operations below, make sure you can answer the following questions:

1. *What does it do*?
2. *What does it return*?

- `begin`
- `set!`
- `for-each`
- `when` and `unless`

Make sure you're also clear on *when you enter the body of each of these operations*. If you don't enter the body, what is returned?

Tutorial 6 Starter Files **(https://bain-cs111.github.io/course-files/tutorials/tutorial_6_template.zip)**

Activity 1 - Basic Imperatives

Let's make a simple bank account simulator!

Activity 1.1

Start by making a global variable, `balance`, and setting it to some initial value. (For the sake of this example we can pretend we have as much money as we'd like instead of being college students)

Activity 1.2

Now make a function `deposit!` that takes a number as input and adds that number to the balance. Have `deposit!` return your resulting balance:

```
; deposit!: number -> number
; Deposits money into our bank account
; Effect: balance increases by deposit number.
```

Hint: You will use `set!` in your implementation. Check the Appendix at the end of Part 1 if you need a refresher.

> **Hint 2**: There is a bit of a trick to returning the updated balance here. Remember `set!` returns a value of (void), so you have to do an extra step to get your deposit function to return a number (you may want to look at `begin` in the documentation).

Activity 1.3

Nice work! Now, write a function called `withdraw!` that takes a number and removes it from the balance, returning the new balance in the process. Your function should only decrease the balance if it is greater than or equal to the amount being withdrawn (we're trying to avoid overdraft fees cause those are bogus).

If for some reason we try to buy something we don't have enough money for, your function should execute:

```
(error "Not enough money!")
```

which will stop the function and print an error message. When you've tested that case, comment out the call to the function that results in the error so that your code will continue to run past that point.

```
; withdraw!: number -> number
; Withdraws money from our bank account!
; Effect: balance decreases by withdraw amount unless
;         balance is less than withdraw number.
```

> **Hint**: You'll want to use `when` to check this condition. Check the appendix below for a refresher

---

Activity 1 Appendix

`set!`

`set!` allows us to modify variables, changing their values after we define them. It is a crucial part of imperative programming. `set!` takes two arguments, a variable and an expression and sets the variable to that expression. So for example:

```
(define hi "hello") ; define some variable
; the variable hi is now "hello"
(set! hi "hola") ; set our variable with a string
; the variable hi is now "hola"
(set! hi (string-append "bon" "jour")) ; a more complex expression
; the variable hi is now "bonjour"
```

`when`

`when` is similar to `if` but an imperative variant. The signature for `when` is `(when question-expression then-expression)`, note there is no `else` expression here.

If the `question-expression` evaluates to `true` then `when` executes the given `then-expression` and returns the result. Otherwise, if the `question-expression` evaluates to `false`, then `when` does nothing and returns `(void)` (this is different than how `if` behaves).

```
(define hi "ni hao") ; define some variable (define bye "goodbye") ; define so
me other variable
(when (string=? hi "hola") (set! bye "so long"))
; hi is set to "ni hao" so this will do nothing and return void
; bye is still set to goodbye
(set! hi "hola") ; set our variable with a string (when (string=? hi "hola")
(set! bye "so long"))
; hi is now set to hola, so we will set bye to "so long"
; bye is now set to "so long" from the when statement
```

`unless`

`unless` is a companion to `when`. Instead of executing the `then-expression` when the `question-expression` evaluates to `true`, *it executes it when the* `question-expression` *evaluates to* `false`. Otherwise all behavior is identical to `when`.

When working in the Advanced Student Language, instead of hitting the **run** button, you can now hit the **debug** button. This will compile the program but not start it yet. It should pop up the **pause**, **go**, **step**, **in**, and **out** buttons (with all but *pause* greyed out), think for a while, and then grey out pause (because it's now paused) and un-grey-out go and step. This will be helpful for the next section of the Tutorial.

The debugger is now waiting for you to either hit **step** or **go**. Before you hit **go**, find an expression you want to "investigate" and hover your mouse over its opening parenthesis. You should see a little red dot appear in the middle of the paren. Right click and choose *"Pause at this point"*. Now, click **go**. It should finish loading the file and give you a prompt in the interaction window. When it reaches the part of the program you set the **breakpoint** at, it will pause and show you the state of the "stack" or what is essentially the computer's memory at that point.

Hit go again, and it will run to the next call of your function. Keep pressing go until it finishes running.

To remove a *break point* right click the same paren you did before and choose *"Remove pause at this point."* Using the debugger allows you to get a peak at the state of the computer in between runs of your imperative functions. This will be very useful for the next part of the assignment.

Activity 2 - Imperative Loops

Now that we've got some basic imperatives down. We need to also use some imperative loops! These are like our iterators from earlier in the quarter, but they behave imperatively.

Activity 2.1

Now write `list-max` using `for-each` and `set!`

```
; list-max: (listof number) -> number
```

> **Hint**: You'll want to use `local` to create a `local` variable that you can modify. Just like `local` can be used to define functions just for use within a function, we can use it to define variables accessible and modifiable only from within that function.

Notice that we don't have an effect statement here because `list-max`, while using imperatives *internally*, won't change any global variables or object fields and so won't have any effects that will be visible outside the function.

(Note: this is very similar to the `sum-list` procedure we did in class. So check out the lecture slides if you aren't sure how to start!)

Activity 2.2

Now rewrite it using iterative recursion (but still using `set!`). Notice that it's a lot more compact with `for-each`.

Activity 3: Simple GUIs

Most of the time in real programming projects, we don't leave our programs in a purely text-based form for someone to interact with. Imagine if you asked a person at an ATM for instance, to interact with your program by typing in specific commands with specific syntax. Most of the time, we separately develop a *front-end* for our programs called a **Graphical User Interface (GUI)** (like the snake game you're working on in Exercise 6!).

At the bottom of the starter file we've included driver code for the simplest possible *text editor* (think Microsoft Word but scaled WAY down). You call the editor by running `(edit-text)`. It will display the text in the string variable `the-text`, which is initially just the empty string (so you won't see any text to begin with).

Each time you press a key, `edit-text` will call the procedure `key-pressed` with the key you typed as an argument, e.g. `"a"`, `"b"`, etc.

Modify `key-pressed` so that it updates the-text with the new character that was typed. You can join two strings using `string-append`, which is just like `append`, but works on strings rather than lists. You can then store that result back in `the-text`.

Removing characters

The current version would be great if humans were infallible, but in practice we make typos. So we need to modify the editor to support backspace. In particular, when the backspace key is pressed, it should remove the character at the end of the-text.

When you press the backspace key (delete on a Mac), the key that gets reported to `key-pressed` is the magic backspace character, which you can't actually type into a string in your source code because as soon as your try, the editor will think you want to erase part of your source code, not that you want to type the backspace character as part of a string. So most programming languages provide a mechanism for typing un-typable characters inside of string constants by using something called an **escape character**, which in most languages is a *backslash* (`\`). The escape character tells the system that the following character should be treated differently than usual. In Racket, a backslash followed by the letter `b` means a backspace character. So if you want to check whether a key is the backspace key, just say `(string=? key "\b")`

Now we need some way of removing characters from a string. The simplest way to do that is to use the `substring` function. If you say `(substring string start end)`, it will return to you the section of string starting at character number `start` (with 0 meaning the first character), up to (but not including) character number `end`.

You want to delete the last character, so you want a substring starting at position `0` and ending at the position of the last character in the string (i.e. the `(- length 1)`. That means you need to know the length of the original string, and you can get that with the `string-length` procedure which takes in

Quitting

Finally, you'd like to have some way of signaling when you're done typing. So modify `key-pressed` to quit the editor when the user presses return/enter. The return key is reported to `key-pressed` as "\r", and you can tell the editor to quit just by setting the variable `quit?` to `true`.

Getting Credit for Your Work

IF YOU ARE IN CLASS OR AT ALT TUTORIAL

Find one other person in your group that is finished and peer review each other's work. Here are the things to check:

1. Does their code look readable and neat?
2. Can you understand what their code does by reading it?
3. How was their solution different from yours (if at all)?

Once you've each taken a look, take a second to debrief. Anything either of you found difficult? Easy? Fun? Mind-blowing? Once you've debriefed, **both of you should fill out this attendance Google Form (https://forms.gle/vhKxTCXHJUtMoD7v8) . NOTE: You will need the NetID of the person's whose code you reviewed.** You do not need to submit your Tutorial RKT file unless you want to.

IF YOU ARE SUBMITTING REMOTELY

You MUST submit your completed tutorial to Canvas. Make sure you have correctly defined all the functions.

Before turning your assignment in, **run the file one last time** to make sure

Availability dates

11/8/2023