Tutorial 5: Imperative programming

Getting started

Start by opening the attached **Tutorial 5.rkt** file.

Now you need to configure DrRacket for the **Advanced Student Language**. Select Choose Language from the Language menu, and select Advanced Student Language. Now you will be able to use imperative programming.

Part 1: Updating a state variable

Make a bank account simulator:

- Start by making a global variable called **balance** and set it to some number.
- Now make a procedure called **deposit!** that takes a number as input and adds it to the balance. Have deposit return your balance as a result.
- Now make a procedure called **withdraw!** that takes a number and removes it from the balance, but only if the balance is sufficient. If not, it should execute: (error "Not enough money!").

Part 2: Imperative loops

Folding

Now write **foldl** using imperatives. You won't be able to redefine the name foldl, so name your procedure something else like my-foldl or imperative-foldl. Your procedure should start by making a local variable to hold the current version of the result. The use **for-each** to run a procedure over the list that updates the result. (Note: this is basically the **sum-list** procedure we did in class, but changed to be foldl rather than always summing. So you can look at the lecture slides for a template.)

Reversing

Write reverse-list as an imperative loop.

Part 3: Imperative tree recursions

Now write **sum-recursive-list** using recursion and an imperative loop. Start by making a local variable to hold the sum. Now write a local helper procedure that, when called on a number, adds it to the sum, and when called on a list, calls itself on every element of the list using for-each. Finally, have sum-recursive list call the helper procedure on its argument and return the sum. So your code should look something like this:

```
(define (sum-recursive-list list)
  (local [(define sum 0)
          (define (helper x) .. fill in…)]
    … fill in…)
```

Part 4: Simple GUI code

We’ve included driver code for the simplest possible text editor. 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.

Adding characters

Modify key-pressed so that it updates the-text with the new character that was typed. You can join to 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.

Escape sequences

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 untypable characters inside of string constants 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 **(equal?** ***key*** **"\b")**.

String surgery

Now we need some way of removing characters from a string. The simplest way to do that is to use the **substring** procedure. If you say (substring *string start length*), it will return to you the section of *string* starting at character number *start* (with 0 meaning the first character), and that’s *length* characters long. You want to delete the last character, so you want a substring starting at position 0 and whose length is one less that the length of the original string. That means you need to know the length of the original string, and you can get that with the **string-length** procedure.

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.