

Lab 04 Specification – Expressions in Other Languages
Due (via your git repo) no later than 8 a.m., Tuesday, 2nd October 2019.
50 points

Lab Goals

- To learn to use prefix and postfix notations for writing expressions.
- To obtain a basic understanding of Scheme/Racket and PostScript programming languages.
- To reflect on the different types of expression notations and outline advantages and disadvantages associated with all.

You are required to work in groups of two in this lab. It is acceptable to have the same team as earlier for this lab. Any team change need to be communicated to the Professor ASAP. Let us keep in mind that developing communication skills within a team environment is very important in the REAL World. So this is a great opportunity for us to develop team skills.

One submission for a team is sufficient.

Once grading is done, the graded report will be uploaded to the submission Git repository. All team members will receive same grade for the lab.

Suggestions for Success

- Take a look at the suggestions for successfully completing the lab assignment, which is available at:
<https://www.cs.allegheeny.edu/sites/amohan/resources/suggestions.pdf>

Learning Assignment

To do well on this assignment, you should also read

- PLP chapter 06, section [6.1 - 6.4]

Tasks

- **[Loading Docker Container.]** There are three steps in loading the container, namely:
 - Build the container
 - Run the container
 - Connect to the container

Build the container: So in order to build the container. the following steps should be performed.

1. First accept the lab url provided in the Slack. After downloading the lab folder from the GitHub classroom, navigate to the cmpsc201-fall-19-lab02 directory using terminal or Docker quick start terminal (windows users).

2. At this point, you should find a directory called myimages inside the parent directory. This can be checked by typing in `ls` command in terminal window. Moving forward in this document, Terminal refers to Docker Quickstart Terminal for windows users.
3. Navigate to the myimages directory using the `cd` command.
4. Build the docker image using the following command:
`docker build -t cs201lab04 .`
5. Note: In the command above, cs201lab02 is the user provided image name. This could be random. But it is recommended to use the same name so as to easily follow the rest of this document. Additionally, it is required to be inside the myimages directory in order to run the build command. If you are not inside the myimages directory, you may receive an error message.
6. Upon successful build, it is recommended to verify the correctness of image creation by using the following command:
`docker image ls`
7. The image named "cs201lab04" should be listed as one of the output from the command above.

Run the container: So in order to create and run the container. the following steps should be performed.

1. Run docker container based on the image created in the previous steps using the following command:
`docker run -t -d -v /Users/amohan/myimages/resources:/usr/share/resources --name cs201container04 cs201lab04`
2. Note: In the command above, replace the directory path "`/Users/amohan/myimages/resources`" with the path of the directory in YOUR machine. Let us suppose you are using a windows laptop, then you may use the following command.
`docker run -t -d -v /c/resources:/usr/share/resources --name cs201container04 cs201lab04`
In the above command, it is required to create the **resources** folder inside the **c** directory.
3. In addition to creating the container, the run command above creates a mount between the host machine and the container for a shared folder space. So, any files placed inside the host mount directory can be easily accessible inside the container mount directory and vice versa.
4. Upon successful run, it is recommended to verify the correctness of container creation by using the following command:
`docker container ls`
5. A container with its ID number should be listed as one of the output from the command above.

Connect to the container: So in order to connect to a running container, we will use the following commands:

1. `docker exec -it cs201container04 /bin/bash`
2. After successful connection, this should log to the container with a root user name.

If you are a lab machine user, you may skip the steps with Docker and directly go to the next steps.

If you are a windows laptop user, you may download and install drracket from the website below:

<https://download.racket-lang.org/>

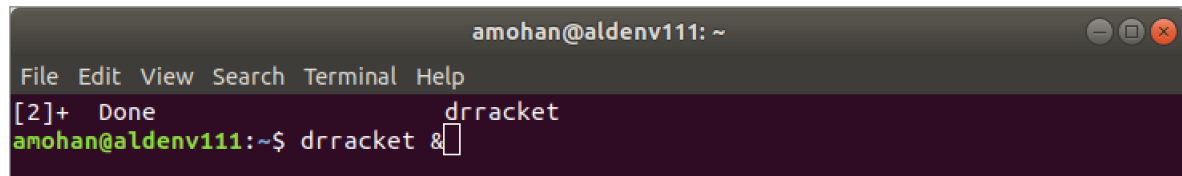
- **[GitHub Setup.]** Take a look at the detailed documentation for getting started with GitHub, which is available at: <https://www.cs.allegheeny.edu/sites/amohan/resources/github.pdf>
- **[1. Evaluate some prefix expressions.]** Racket is a dialect of Scheme, which in turn is an extension of a very old language called LISP. We have an environment devoted entirely to Racket, called "DrRacket".

If you are a linux/mac user, you can access it from the docker container. In the terminal, after connecting to the container just type `racket`. This would open a command line interface, in which the LISP commands can be executed. The command below, will exit from the DrRacket interface:

(exit)

If you are a windows user, it is recommended to watch the YouTube link provided below. The video provides a detailed demo of how to access DrRacket. <https://www.youtube.com/watch?v=UOqcLGGKvr8>

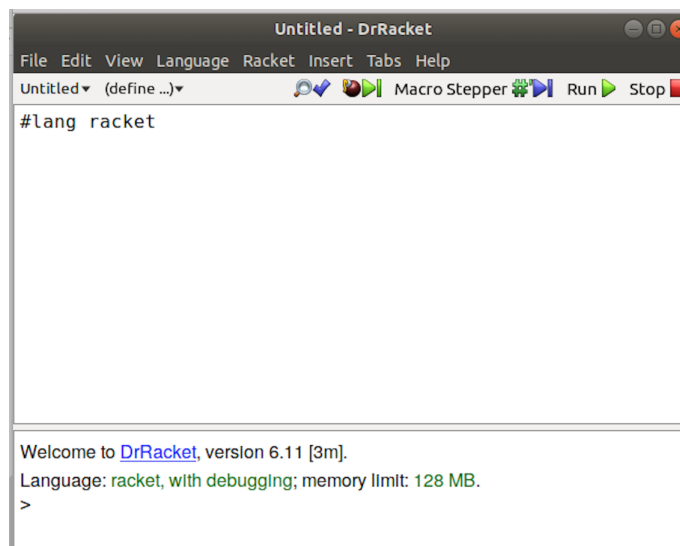
If you are a lab machine user, to access it, just type `drracket &`, similar to the screenshot displayed below:



```
amohan@aldenv111: ~  
File Edit View Search Terminal Help  
[2]+ Done drracket  
amohan@aldenv111:~$ drracket &
```

Make sure the top frame of the window shows the words “# lang racket”—if not, enter that line. You should see two frames; the bottom one should say “Welcome to DrRacket.” Expressions entered in the top are gathered together and executed as a batch when you hit the CTRL-R key. Expressions entered in the bottom are evaluated immediately.

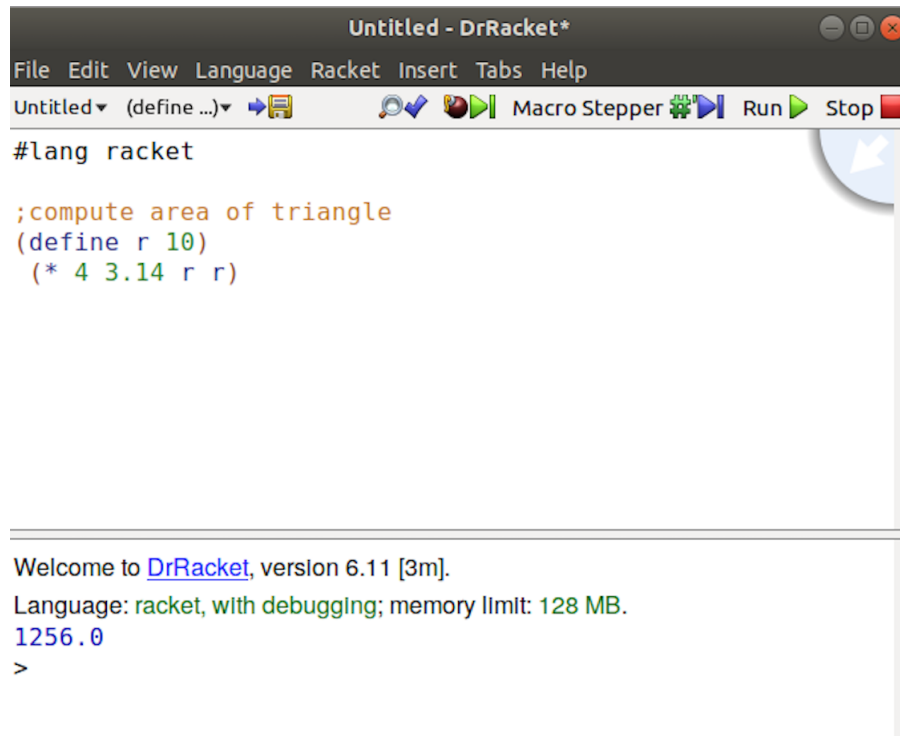
The DrRacket pop up window, should look similar to the screenshot shown below:



Study the tutorial provided on <https://docs.racket-lang.org/quick/> and try the examples discussed in the tutorial. You may also use the complete documentation as a reference: <https://docs.racket-lang.org/guide/>.

Take a look at the examples in next page, for some sample racket code written by me:

The example below calculates the area of a triangle:



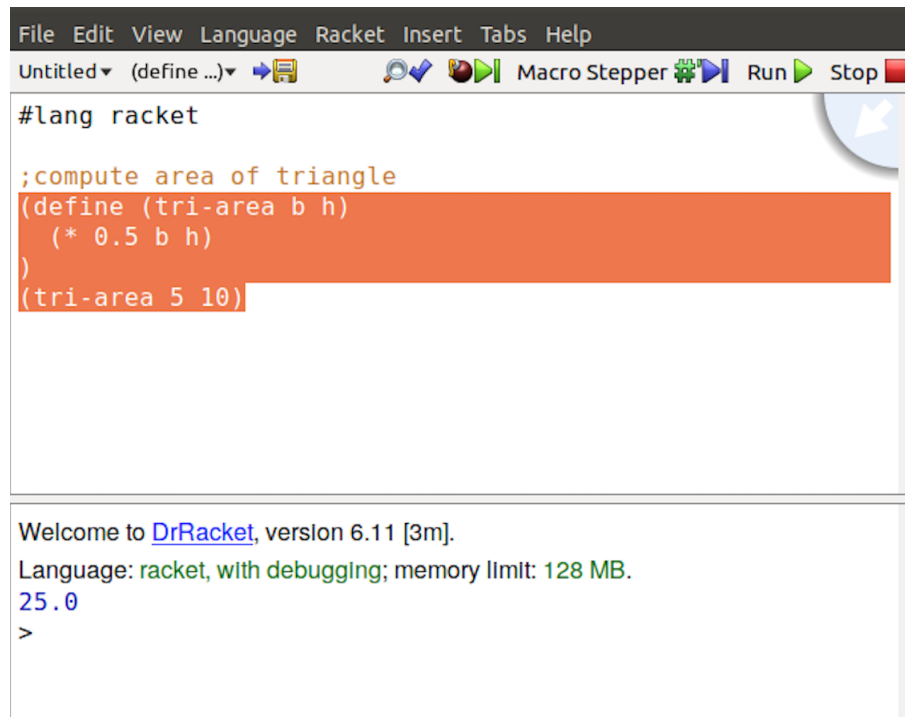
The screenshot shows the DrRacket IDE window titled "Untitled - DrRacket*". The menu bar includes File, Edit, View, Language, Racket, Insert, Tabs, and Help. The toolbar contains icons for saving, undo, redo, and running code. The code area contains the following Racket code:

```
#lang racket

;compute area of triangle
(define r 10)
(* 4 3.14 r r)
```

The bottom panel shows the welcome message: "Welcome to [DrRacket](#), version 6.11 [3m]. Language: racket, with debugging; memory limit: 128 MB. 1256.0 >

The example below calculates the area of a triangle using a function called tri-area:



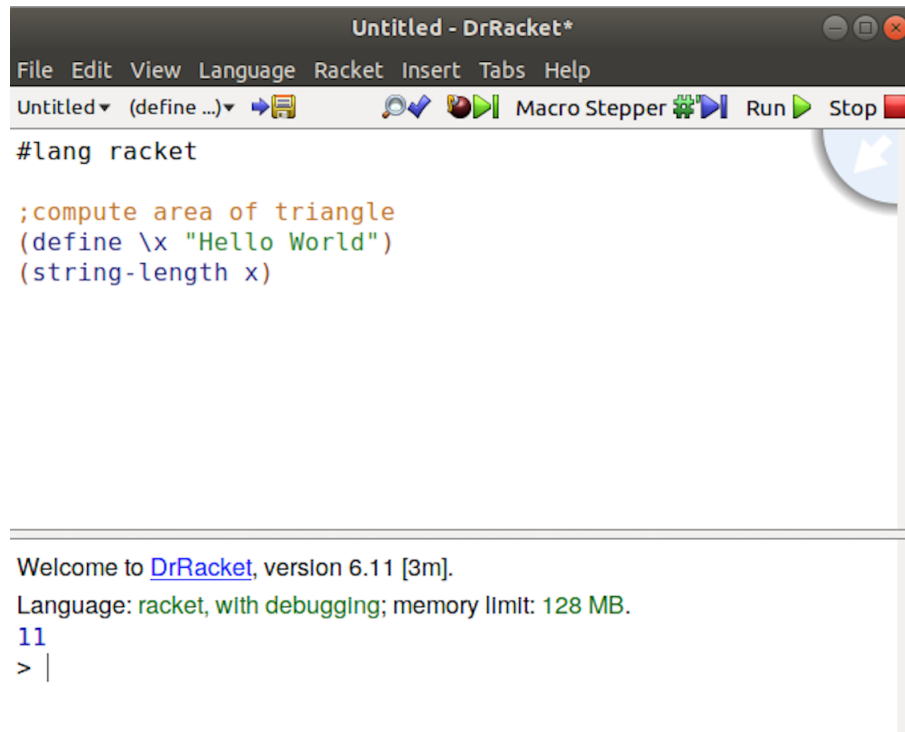
The screenshot shows the DrRacket IDE window titled "Untitled - DrRacket*". The menu bar includes File, Edit, View, Language, Racket, Insert, Tabs, and Help. The toolbar contains icons for saving, undo, redo, and running code. The code area contains the following Racket code:

```
#lang racket

;compute area of triangle
(define (tri-area b h)
  (* 0.5 b h)
)
(tri-area 5 10)
```

The bottom panel shows the welcome message: "Welcome to [DrRacket](#), version 6.11 [3m]. Language: racket, with debugging; memory limit: 128 MB. 25.0 >

The example below calculates the length of the given string:



The screenshot shows the DrRacket IDE window titled "Untitled - DrRacket*". The menu bar includes File, Edit, View, Language, Racket, Insert, Tabs, and Help. The toolbar contains icons for saving, undo, redo, running, and stopping. The code editor contains the following Racket code:

```
#lang racket

;compute area of triangle
(define \x "Hello World")
(string-length x)
```

Below the code editor, the DrRacket welcome message is displayed: "Welcome to DrRacket, version 6.11 [3m]. Language: racket, with debugging; memory limit: 128 MB." The prompt shows the line number 11 and a greater-than sign followed by a vertical bar, indicating the current cursor position.

After you have studied some Racket code, create a new Racket file and enter code for the following tasks (precede each one with a comment line giving the problem number, e.g., "1(a)"). If a problem calls for a symbolic name (e.g., "x"), use "(define ...)" to create the name and associate it with a value.

1. Define a symbol `r` with value 10, then write a one-line expression describing the surface area of a sphere of radius `r` (you can look this up).
2. Using your symbol `r` from the previous part, write a one-line expression describing the volume of a sphere of radius `r` (you can look this up; calculate the "4/3" in Racket rather than using a decimal approximation; don't use the `expt` function).
3. Define symbols for `a`, `b`, `c`, and `x` with values 1.2, 2.3, 3.4, and -2, respectively. Write a one-line expression that finds the value of $ax^2 + bx + c$.
4. Define a symbol `s` with value "Hello, Racket". Write a one-line expression that defines a symbol `mid` equal to half the length of `s`, rounded down to the nearest integer. (Use the `string-length` function.) Actually calculate this in your expression, don't just enter a constant! Display the value of `mid` (i.e., just write the expression `mid`).
5. Using the symbols `s` and `mid` that you defined previously, write a one-line expression using `substring`, `string-append`, and the constant string "Dr. " that creates the string "Hello, Dr. Racket".
6. Use `define` to create a function named `area` that has one argument, a radius `r`, and computes the area of a sphere of radius `r`. Evaluate your function with `r` equal to 10, 20, and 30.
7. Use `define` to create a function named `vol` that has one argument, a radius `r`, and computes the volume of a sphere of radius `r`. Evaluate your function with `r` equal to 10, 20, and 30.
8. Use `define` to create a function named `midpt` that has one argument, a string `s`, and computes half the length of `s`, rounded down to the nearest integer. Evaluate your function with the strings "a string", "dr. racket", and "abcde".

9. **[Optional.]** Use `define` to create a function named `insert` that has two arguments `s` and `t`, both strings, and creates a string consisting of `t` inserted into the middle of `s`. Test it with several examples of your own choosing. (Use the function you created in the previous part.)

Make sure you have commented all portions of your program and save your Racket file in your repository.

• **[2. Evaluate some postfix expressions.]**

For this part, there is no need to use Docker. Any text editor such as Atom and a PostScript viewer is sufficient to implement the functionality in this part.

The repository has a few sample PostScript files. If you are a linux user, to run them, type `evince` followed by the file name, e.g., “`evince ex1.ps`” or just open them from any PostScript viewer.

For example, in MAC, I use `preview` to run the files. If you are a windows user, please follow the link below to install a PostScript viewer: <https://kb.iu.edu/d/afbn>

Study the code of these file (by opening them with a text editor). There are many online resources with Postscript examples and you may find a complete PostScript reference on:

<http://www.adobe.com/devnet/postscript.html>.

The commands you will use today are:

- `x y moveto` – move the pen to location (x, y)
- `rx ry rmoveto` – move the pen `rx` in the x -direction, `ry` in the y -direction. In other words, move *relative to* the current position. If the pen is at $(100, 200)$, then “`20 30 rmoveto`” will place it in location $(120, 230)$.
- `x y lineto` – draw a line to location (x, y)
- `rx ry rlineto` – draw a line `rx` in the x -direction, `ry` in the y -direction (in other words, draw *relative to* the current location)
- `x y add` – same as $x + y$
- `x y sub` – same as $x - y$
- `/x expression def` – assign $x = \text{expression}$
- `stroke` – When you complete a figure, this “strokes the ink” onto the page; it does not change the location of the pen
- `showpage` – should be the last thing in your file.

Create two PostScript files that use NO OTHER POSTSCRIPT COMMANDS besides the ones described above. The first file should draw at least four different figures (not rectangles or triangles) in four different places on the page. Comments in your program should describe these (e.g., “%Draws an ‘F’ shaped polygon in the upper right corner”); further comments are welcome, of course.

The second file should be a simple composition of your own devising. It should use simple arithmetic expressions. It should have at least four distinct components (which could be shapes or lines). In your comments you may make an artistic statement about your composition.

- **[3: Self Reflection]** Self Reflection is important to any learning environment. I like you to spend some time thinking about what you had learned in this lab and how it correlates to the class discussions we did together. Add a Reflection document (in PDF format) to the repository. The Self reflection should be as detailed as possible. If the self reflection is not detailed and let us say if you had just included one or two lines superficially then points shall be deducted for this part.
 - List out your view of using various types of notation (infix/prefix/postfix), discuss advantages and disadvantages of each one.
 - List out the reflections on the biggest learning points and any challenges that you have encountered during this lab.
 - Provide details on how the team work was conducted during the lab and describe the tasks each member completed (if tasks were completed separately). Only one submission is required from a team.
 - List out the general sentiments regarding the lab, such as the likes and dislikes through outlining a solid reasoning behind your sentiment

Required Deliverables

Please submit electronic versions of the following deliverables to your teams Submission GitHub repository by the due date:

1. A properly completed and commented source programs.
2. A reflection document in PDF format as outlined above.

Grading Rubric

1. If you complete Task 1 completely as per the requirement outlined above, you will receive 25 points.
2. If you complete Task 2 completely as per the requirement outlined above, you will receive 20 points.
3. If you complete the Reflection document as per the requirement outlined above, you will receive 5 points.
4. If you fail to upload the lab solution file to your git repo by the due date, there will be no points awarded for your submission towards this lab assignment. Late submissions will be accepted based on the late submission policy described in the course syllabus.
5. Partial credit will be awarded, based on the work demonstrated in the lab submission file.
6. If you needed any clarification on your lab grade, talk to the instructor. The lab grade may be changed if deemed appropriate.

