Lab 03 Specification – Expressions in Other Languages
Due (via your git repo) no later than 8 a.m., Tuesday, 25th September 2018.
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. So find a team member to work with.
May be you could reconnect to your team mate, with whom you partnered for the presentation given on programming languages earlier this semester.
If you needed a team member, ask in Slack. Let us connect to each other and learn from each other!
Make sure to fill out the google form (link provided to your allegheny email account) by tomorrow morning 8.00 am, with your team details.
The Submission GitHub account is required to be filled out, which is one of your team members Git account.
The repository associated with the submission Git account provided, would be used for grading your team's work.
Once grading is done, the graded report will be uploaded to the submission Git repository provided in google form and 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.allegheny.edu/sites/amohan/resources/suggestions.pdf`

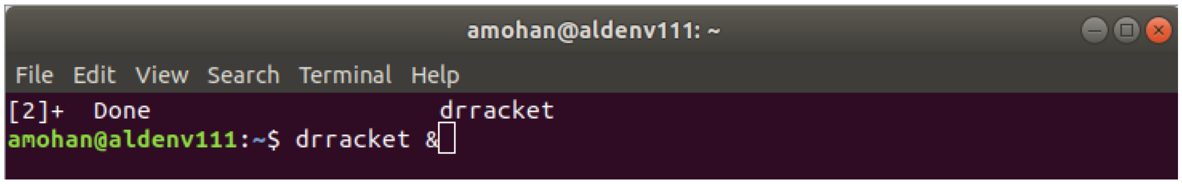
Learning Assignment

To do well on this assignment, you should also read

- PLP chapter 06, section [6.1 - 6.4]

Tasks

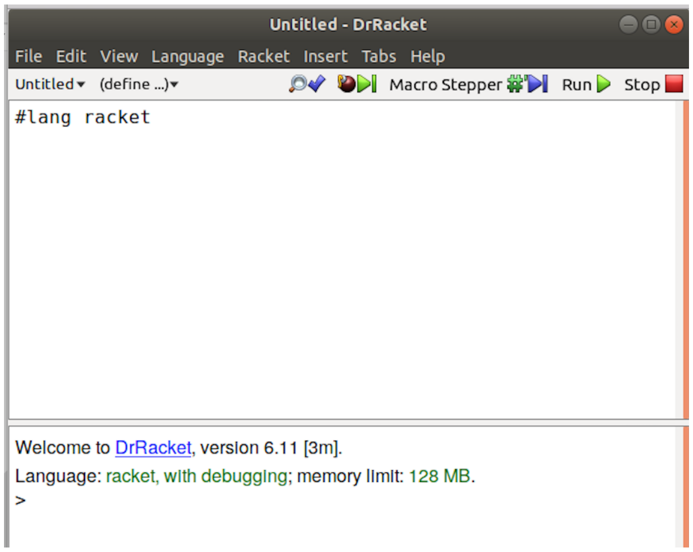
- **[1. Evaluate some prefix expressions.]** Racket is a dialect of Scheme, which in turn is an extension of a very old language called LISP. We have an environment devoted entirely to Racket, called "DrRacket". To access it, just type `drracket &`, similar to the screenshot displayed below:



```
[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:



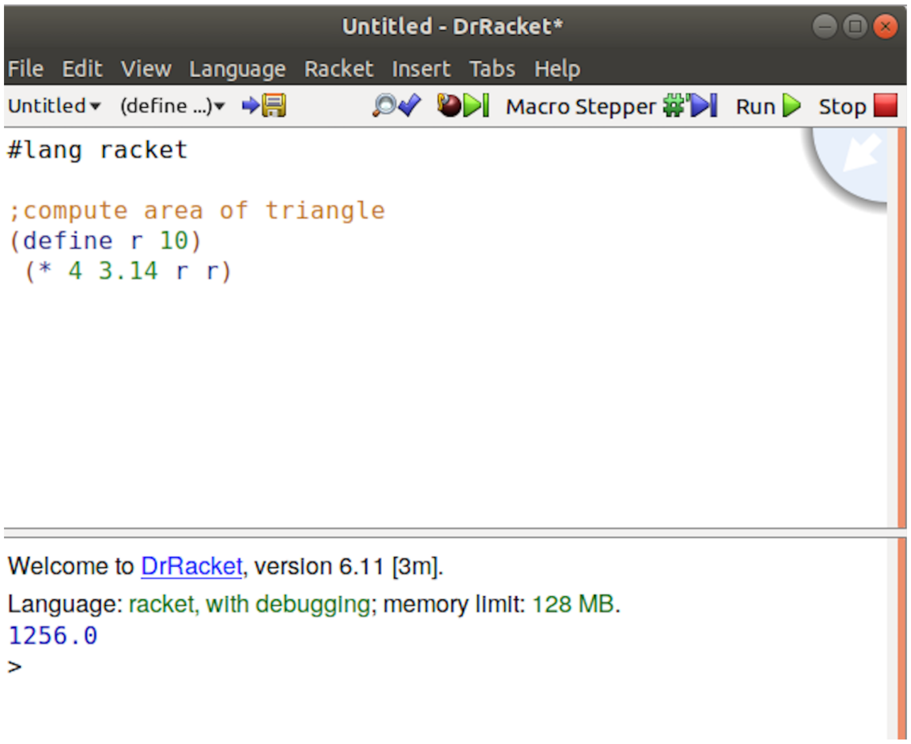
```
#lang racket
```

```
Welcome to DrRacket, version 6.11 [3m].
Language: racket, with debugging; memory limit: 128 MB.
>
```

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 following, for some sample racket code written by me:

The example below calculates the area of a triangle:

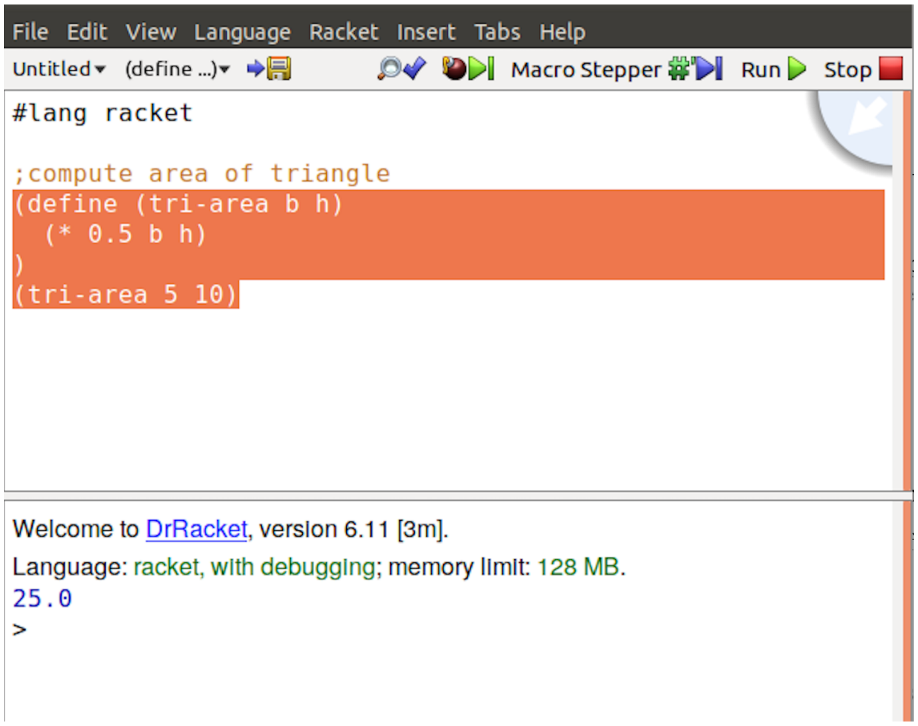


```
#lang racket

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

```
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:

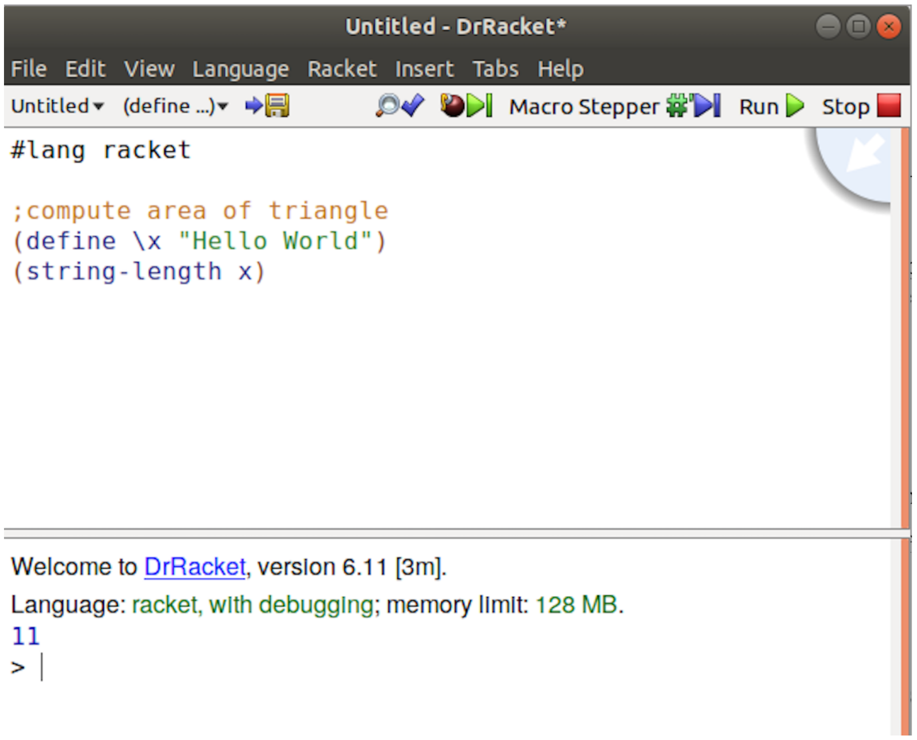

File Edit View Language Racket Insert Tabs Help
Untitled (define ...) Macro Stepper Run Stop

```
#lang racket

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

```
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:


Untitled - DrRacket*
File Edit View Language Racket Insert Tabs Help
Untitled (define ...) Macro Stepper Run Stop

```
#lang racket

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

```
Welcome to DrRacket, version 6.11 [3m].
Language: racket, with debugging; memory limit: 128 MB.
11
>
```

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 `r`. 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.]** The repository has a few sample PostScript files. To run them, type `evince` followed by the file name, e.g., "`evince ex1.ps`" or just open them from GUI. 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 $x = expression$
  - `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.

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 with your view of using various types of notation (infix/prefix/postfix), discuss advantages and disadvantages of each one, and also provide reflections on the biggest learning points and any challenges that you have encountered during this lab.
3. In your reflection document, you should also 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.

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 15 points.
3. If you complete the Reflection document as per the requirement outlined above, you will receive 10 points.
4. If you fail to fill out the google form by tomorrow 8.00 am, your work will not be graded and there will be no points awarded for your submission towards this lab assignment.
5. 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.
6. Partial credit will be awarded, based on the work demonstrated in the lab submission file.
7. If you needed any clarification on your lab grade, talk to the instructor. The lab grade may be changed if deemed appropriate.