

This lab introduces you to the very basics of the **DrScheme** programming environment, which we will be using throughout the course. *You should work alone on this lab.* Also, see the course homepage for the due date.

§ 1. Running DrScheme for the first time

On the ECS Unix systems, you can invoke **DrScheme** by typing `drscheme` at the Unix prompt.

Once **DrScheme** has started up, you will need to select the correct language level, as follows:

From the **Language** menu, select **Choose Language...**, and then select **Essentials of Programming Languages (3rd Edition)**.

Selecting the language is a one-time-only process: you should not have to repeat these steps the next time you start up **DrScheme**.

§ 2. Defining procedures

Notice that there are two frames to the default environment. The top one is for editing programs, while the bottom one is for interacting with your programs. Type the following procedure definition into the editing (i.e., top) frame:

```
(define simple (lambda (a b) (+ (* a 100) b)))
```

Note that **DrScheme** paren-balances the expression as you type it. The rough equivalent function definition in Java would look something like:

```
public static int simple(int a, int b) {
    return ((a*100) + b);
}
```

The rough equivalent Haskell function (omitting the type declaration) would be either one of the following:

```
simple a b = (a*100) + b
simple = \ a b -> (a*100) + b
```

There are two main ways to send expressions to the interpreter (i.e., the lower frame):

1. You can interact with the interpreter directly. For example, type

```
(* (+ 1 2) (- 3 6))
```

at the prompt ("`>`"), hit return, and the interpreter should respond with the value `-9`.

You will also get a warning message indicating that there are definitions in the editing frame that haven't yet been loaded into the interpreter. Don't worry, we're about to take care of that.

2. You can click the "**Run**" button, which will load the entire contents of the editing frame into the interpreter.

Click the "**Run**" button, and the definition of `simple` will be sent to the interpreter. Now, in the interpreter frame, type

```
(simple 4 6)
```

and hit return. The value `406` should appear as a result.

Also evaluate the following:

```
simple
```

Notice that the result is:

```
#<procedure:simple>
```

This is the system's way of saying that the value of `simple` is a procedure whose internals you cannot see.

Now, type the following into the editing frame:

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

Note that your lines are automatically indented for you, to help keep your code legible and reasonably formatted. You can send this definition to the interpreter by clicking **Run** again. This is the usual recursive definition of the factorial function. In Java it would look something like:

```
public static int fact(int n) {
  if (n==0)
    return 1;
  else
    return (n*fact(n-1));
}
```

Evaluate from the interpreter:

```
(fact 3)
```

Also evaluate:

```
(fact 100)
```

(Try that with your favorite C or Java compiler!) Note that the definition of `fact` has a standard pattern for a numeric recursion:

```
(lambda ( n ... other parameters ... )
  (if (zero? n)
      some value ; the "then" clause
      some expression with a recursion ; the "else" clause
  ))
```

§ 3. Checking syntax

Click the **Check Syntax** button, and then run your cursor over the definition of `fact` in the editing frame. You will see small arrows that illustrate the relationship between variables in the code and their defining occurrences.

For example, if you position the cursor over the `n` in `(zero? n)`, you will see an arrow from the `n` in `lambda (n)`.

If, instead, if you position the cursor over the `n` in `lambda (n)`, you will see arrows to all occurrences of variables that correspond to that definition.

§ 4. Saving, editing, and running your work

Even though you've loaded definitions into the interpreter, you have not saved the definitions to a file.

Return (if you're not already there) to your editing frame, and delete everything except for the definition of `fact`. To save this definition, click the **Save** button near the top of the window, and you will be prompted for a location to save the file. (In the instructions that follow, I'll assume that you saved them to the file `fact.ss` in your current working directory.) If you make additional changes to the file that you want to save, simply click **Save** again: you will not be prompted for a file name again.

Once again, click the **Run** button. Only the definitions in the editing frame (i.e., the definition of `fact`) will exist in the interpreter: the definition of `simple` no longer exists. To verify, type the following at the prompt:

```
simple
```

You will get an error message indicating that `simple` is an undefined identifier. In fact, each time you click Run, a fresh interpreter is initiated, using only the definitions in the editing frame. Note that you can go back to the editing frame, make changes (with or without saving them), and send the revised expressions to the interpreter to be evaluated. However, it is a good idea to save your file as you work, so that you don't accidentally lose important information.

§ 5. Basic debugging

In the interpreter, first evaluate

```
(trace fact)
```

and then

```
(fact 4)
```

You should see a trace of the evaluation of `(fact 4)`. To turn off tracing on `fact`, evaluate

```
(untrace fact)
```

Do this before you go on. Now click the **Debug** button: you will still have two frames, but there will be some new buttons near the top of the window (**Pause**, **Continue**, and **Step**).

In the interpreter (i.e., lower frame), again evaluate the following:

```
(fact 4)
```

In the upper frame, you will see a small arrow pointing at the **if** in **fact**. Click **Step** two times, and the arrow will be on the closing parenthesis in the expression **(zero? n)**. In the upper left, you'll see text indicating that this expression is evaluating to **#f**; if you move your cursor over the **n** in the code, you'll see (to the right of the **Step** button) that the current value of **n** is **4** (i.e., the argument that was initially passed into **fact**). Click **Step** a few more times to see how the debugger steps through the code. When you get sufficiently bored, click **Continue**, and the interpreter will complete its evaluation. At this point, if you type in the upper frame, **DrScheme** immediately returns you to the editing frame.

§ 6. Including test cases

Here is a simple way of incorporate test cases into your definitions file. (There are fancier ways of doing this, but we will worry about them later in the course.) I'll use the test case of **(fact 5)** (which I know should evaluate to $5 \times 4 \times 3 \times 2 \times 1 = 120$) as an example:

Insert at the bottom of the definitions specific calls, along with the anticipated answer. For example, add the following to your **fact.ss** definitions:

```
"(fact 5) is"
(fact 5)
"should be"
120
```

When you click **Run**, you should see the following in the interpreter window:

```
"(fact 5) is"
120
"should be"
120
```

If for some reason the two answers don't match up, you can then spend some time determining whether your code is incorrect or whether you simply miscalculated what the right answer should be.

§ 7. Your problem

Define a Scheme procedure that computes the function **sum-cubes** such that, for each integer n , we have

$$\text{sum-cubes}(n) = \begin{cases} 0, & \text{if } n \leq 0; \\ n^3 + \text{sum-cubes}(n - 1), & \text{otherwise.} \end{cases}$$

Test it out on some small values, including at least one negative number. Use one of the methods from §6 for defining your test cases.

Notes

- ▶ To do the $n \leq 0$ test in Scheme, write **(<= n 0)**.
- ▶ Also, ***** is like **+** in that it takes arbitrarily many parameters, i.e., **(* 1 2 3 4)** returns **24**.

What to hand in: Hand in the source code for **sum-cubes**, along with verified test cases that demonstrate that the code works as required. You do not have to submit anything electronically for this lab. You should submit it to the labeled bin in CST 3-212.