Compilers

Lab Session 2

Assistants: Antoine Van Muylder & Sander Huyghebaert

The goal of this exercise session is familiarizing you with the libraries that will be used throughout the projects.

1 Libraries

This section serves as documentation for the libraries mentioned. You will need to be familiar with these in order to make the exercises.

1.1 Racket Parameters

Before we dive into the libraries provided in the compilers book, we will first take a look at *parameters*[1] in Racket. No additional `require` is necessary, *parameters* are part of the `racket/base` package, which is imported by default when you write `#lang racket` at the top of your file.

Parameters essentially allow us to use *dynamic bindings*. We can create a parameter using the `(make-parameter initial-value [guard name])` parameter (*guard* and *name* are optional parameters, this is denoted by the square brackets). For more information on the *guard* and *name* parameters, refer to the documentation.

Let's define our first parameter:

```
(define name (make-parameter "John Doe"))
```

This binds *name* to the parameter procedure created by `make-parameter`. We can retrieve the contents of the *name* parameter by calling it:

```
name   ;; => parameter procedure
(name) ;; => "John Doe"
```

We can use it in procedures like so:

[1] `https://docs.racket-lang.org/reference/parameters.html`

```
(define (greet)
  (print (string-append "Hello " (name))))

(greet) ;; => "Hello John Doe"
```

We can change the value of a parameter temporarily using `parameterize`:

```
(parameterize ([name "Edsger W. Dijkstra"])
  (greet))
;; => "Hello Edsger W. Dijkstra"

(greet) ;; => "Hello John Doe"
```

Parameters are used throughout the milestones:

- for the available registers;
- for the current pass list of the compiler;
- ...

1.2 Info Library

The first library of the compilers book we will look into is the *info* library, which can be imported using `(require cpsc411/info-lib)` (given that you followed all the steps in the project A pdf for adding the cpsc411 package). An *info* is an association list in which each key must be mapped to a proper list. The difference is that an *info* is printed prettier, but it uses slightly more memory:

- *info*: `((key value) ...)`
- *assocation list*: `((key . value) ...)`

You only need a few procedures to work with *info*'s:

- `(info? v)`: predicate that returns `#t` if v is an *info*;
- `(info-ref info key [default])`: returns the value associated with *key* in *info*, if the key doesn't exist, an error is raised, unless you specify a *default* value;
- `(info-set info key value)`: maps *key* to a list containing *value* for *info*, note that it returns the updated info, the passed *info* is not mutated;
- `(info-remove info key)`: removes *key* from *info* and returns the updated info result.

These procedures are implemented using the Dictionaries[2] package, which is possible because the *info* datatype is an association list.

The following code shows how to use an *info* for the inventory of a store:

```
(define inventory (info-set
                   (info-set empty 'apples 2)
                   'oranges 10))
inventory                          ;; => '((apples 2) (oranges 10))
(info? inventory)                  ;; => #t
(info-ref inventory 'apples)       ;; =>  2
(info-ref inventory 'bananas)      ;; => ERROR no value for key bananas
(info-ref inventory 'bananas 0)    ;; => 0
(info-remove inventory 'oranges)   ;; => '((apples 2))
(info-set inventory 'apples 1)     ;; => '((apples 1))
```

1.3 Graph Library

The graph library you will be using represents graphs as an association list from vertices to a list of vertices and can be imported using `(require cpsc411/graph-lib)`. A directed edge exists between a vertex *V1* to *V2* if *V2* is in the list associated with *V1*. An undirected edge exists between *V1* and *V2* if there exists a directed edge between *V1* and *V2*, and between *V2* and *V1*.

1.3.1 Creating a Graph

Using the `new-graph` procedure, you can create a graph. This procedure accepts an optional argument, *vs*, a list of vertices to include in the graph. There is also a predicate available to check if something is a graph, `graph?`.

Some examples:

```
(new-graph)           ;; new, empty graph
(new-graph '(a b c)) ;; new graph with vertices a, b and c
(graph? (new-graph)) ;; => #t
```

1.3.2 Vertices

You can add a vertex *v* to a graph with the `add-vertex` procedure:

```
(add-vertex (new-graph) 'v)   ;; returns the new graph with vertex v
(new-graph '(v))              ;; same as above, but shorter
(define g (new-graph))        ;; new empty graph
(define ng (add-vertex g 'v))
;; ng is bound to the updated graph with vertex v
```

[2] https://docs.racket-lang.org/reference/dicts.html

Removing a vertex v can be done with `remove-vertex`:

```
(define g (new-graph '(v w)))
(define ng (remove-vertex g 'v))
;; ng is bound to the updated graph in which vertex v has been removed
```

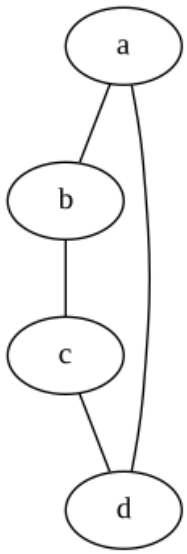
1.3.3 Edges

There are four procedures available to add edges between vertices, one for directed edges, one for undirected edges and a variant on those two to add multiple edges at once:

- `(add-directed-edge g u v)`
- `(add-edge g u v)`
- `(add-directed-edges g u vs)`
- `(add-edges g u vs)`

These procedures take as first parameter a graph g, a source node u and either a single target node v or a list of target nodes vs, for the (directed) edge(s).

A simple example of an undirected graph:



This graph can be created with the following Racket code:

```
(define example-graph
  (add-edge
    (add-edge
      (add-edges (new-graph '(a b c d)) 'a '(b d))
      'b 'c)
      'c 'd))
```

1.3.4 Neighbors

The `get-neighbors` procedure returns a list of vertices that have an incoming edge from the given vertex:

```
(get-neighbors example-graph 'a) ;; => '(d b)
```

2 Exercises

2.1 Parameterized Counter

Create a parameterized `counter` and `step` variable, initialized to 0 and 1 respectively. Write a `get` procedure to retrieve the value of the counter and an `increment` procedure to increment the counter by the value of `step`.

The following tests should pass:

```
(module+ test
  (check-equal? (get) 0)
  (increment)
  (check-equal? (get) 1)
  (parameterize ([counter 100]
                 [step     10])
                 (check-equal? (get) 100)
                 (increment)
                 (check-equal? (get) 110)
                 (check-equal? (counter) 110)))
```

2.2 Info Needed

In this exercise you need to replace all `(TODO)`'s with an expression that will make the tests pass, using the procedures of the *info* library:

```
(module+ test
  (require cpsc411/info-lib)

  (check-equal? (info? '()) (TODO))
  (check-equal? (info? '((apples . 10))) (TODO))
  (check-equal? (info? '((apples 10))) (TODO))
```

```
(check-equal? (info-ref (TODO) 'students) 120)

(let ([data (TODO)])
  (check-true (info? data))
  (check-equal? (info-ref data 'compilers) 'thursday)
  (check-equal? (info-ref data 'hop) 'monday)
  (check-equal? (info-ref (TODO) 'hop) 'never)
  (check-false (info-ref (TODO) 'hop #f))))
```

2.3 Extract Vertices

Given that graphs are represented as dictionaries, can you find a procedure that extracts the vertices from a graph as a list? Hint: take a look at the dictionaries procedures at `https://docs.racket-lang.org/reference/dicts.html`.

Fill in the name of the procedure in the following test (on the `(TODO)`):

```
(module+ test
  (require racket/dict
           cpsc411/graph-lib)

  (check-equal? ((TODO) example-graph) '(d c b a)))
```

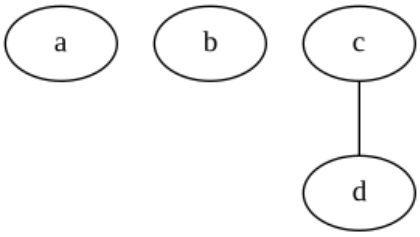
2.4 Subgraphs

Write a predicate, `(subgraph? g1 g2)`, that returns true if g1 is a subgraph of g2. A graph $G_1 = (V_1, E_1)$ is a subgraph of $G_2 = (V_2, E_2)$ if, $\emptyset \neq V_1 \subseteq V_2$ and $E_1 \subseteq E_2$ (where V_1 and V_2 are the set of vertices of a graph and E_1 and E_2 are the edges of a graph).

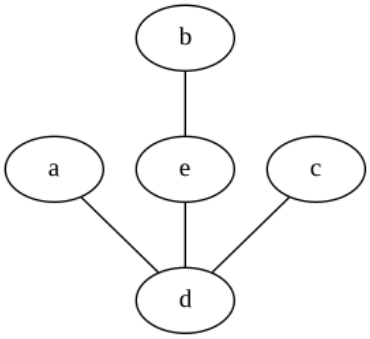
Here are some tests to see if your implementation is correct:

```
(module+ test
    (check-true (subgraph? (new-graph '(v)) (new-graph '(v))))
    (check-false (subgraph? (new-graph) (new-graph '(v))))
    (check-true (subgraph? (add-edge (new-graph '(v w)) 'v 'w)
                           (add-edges (new-graph '(v w x)) 'v '(w x)))))
```

In the following test snippet, complete the definitions of g_1 and g_2 so that the test can be run and passes.



g1



g2

```
(module+ test
    (define g1 (TODO))
    (define g2 (TODO))
    (check-true (subgraph? g1 g2)))
```

2.5 Outputting Graphs

For this exercise, you will write a procedure that translates a graph to the DOT[3] language. The signature of the procedure is `(print-graph graph)` and should just *display* the DOT language output (you should use display to avoid printing a string as "Hello", display will print it as Hello, without the double quotes)[4]. You can assume that all graphs passed to `print-graph` are *directed* graphs.

The following DOT code represents a directed graph version of g_1:

```
digraph { // digraph means that the graph we are declaring is a directed graph
  a;      // declare a vertex a
  b;      // declare a vertex b
  c;      // declare a vertex c
```

[3] https://graphviz.org/doc/info/lang.html

[4] See https://docs.racket-lang.org/guide/read-write.html

```
  d;       // declare a vertex d

  c -> d; // an edge from c to d
  d -> c; // an edge from d to c
}
```

Because you are using a Racket procedure to `display` output, you can control where that output is written to because it uses a *parameter* called *current-output-port*. The easiest way to *parameterize* this is using `with-output-to-file`, so that you don't need to worry about closing the file port yourself:

```
(with-output-to-file "graph.dot" #:exists 'truncate
  (lambda () (print-graph g1)))
```

You can then view the graph using a dot viewer. An example dot viewer is *xdot*:

```
$ xdot graph.dot
```

The output should be similar to that of the image of g_1 used in this document. As an optional exercise you could modify your (or add a new) procedure to print undirected graphs, to avoid having two edges between each vertex that has an undirected edge to another vertex.

Your `print-graph` procedure might come in handy if you run into issues when writing the register allocator and wish to see the graph you are working with.