Compilers

Lab Session 1

Assistants: Antoine Van Muylder & Sander Huyghebaert

In this exercise session we aim to write a first compiler SRC → TGT where SRC is a language of simple arithmetic expressions and TGT is a stack machine. Please read each section carefully as we will introduce concepts from Racket you might not be familiar with.

Questions? Ask your questions:

- during the lab sessions
- using canvas
- sending an email

You can reach Antoine Van Muylder at `antoine.van.muylder@vub.ac.be` or using this phone number: `+32 472 69 10 22` (sms, whatsapp). There are no stupid questions.

1 Some prerequisites

1.1 Modules

Each file is its own module (although you can specify submodules, for example for tests) and exporting definitions of a module is done by *providing* them. The following `provide` expression exports all top-level definitions available in your file:

```
;; FILE: greeter.rkt
#lang racket ;; we will be working in the Racket language

(provide (all-defined-out))

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

If you then have another file (for this example we will assume this is a file in the same directory), you can import it like so:

```
;; FILE: main.rkt
#lang racket

(require "./greeter.rkt")

(greet "John Doe")
```

If you were to run the `main.rkt` file with `$ racket main.rkt` it would print "Hello John Doe". For more information, go to the Modules[1] section of the Racket documentation.

[1] `https://docs.racket-lang.org/guide/modules.html`

1.2 Quoting

Racket proposes a concise syntax to easily describe nested lists: the quote notation.

```
(define matrix '((M00 M01 M02) (M10 M11 M12) (M20 M21 M22))  )
(define code '(begin (assign x 10) (assign z #t))  )
(define empty-list '() ) ;;FYI   '()  =  empty
(define sm-cons '(1 2 . (3 4))  ) ;; => as '(1 2 3 4).
```

As showcased in the second example this feature will in fact turn out to be crucial to write structured sentences of embedded programming languages. By default, if a datum (element appearing in a quote form) is not a boolean, a number, a string, etc ..., it is parsed as a symbol. Note that lists can also be specified using alternative functions:

```
(list 1 2 3) ;; => '(1 2 3)
(cons 1 (cons 2 empty)) ;; => '(1 2)
```

Datums appearing in a quote form are always considered as constants by racket. Consequently, we will have to use **quasiquoting**[2] (with the backtick `) in order to introduce variables in quote forms. The said variables can be unquoted with a comma. For instance:

```
(define var 47)
`,var  ;; => to 47
`(1 2 3 ,var) ;; => to '(1 2 3 47)
(define alist `(4 ,(+ var 3))  )
;;unquote inside quasiquote "`" with the comma ","
```

The last expression `alist` evaluates to `'(4 50)`.

Finally we will regularly use *unquote splicing* (denoted `,@`), an additional feature of quasiquoting. Unquote splicing allows you to unquote a list and to insert all of its elements (say n "things") in the expression you are building rather than just inserting the list itself (1 "thing").

```
(define upper-half '(5 6 7 8))
`(1 2 3 4 ,@upper-half) ;; => '(1 2 3 4 5 6 7 8)
`(1 2 3 4 ,upper-half) ;; => '(1 2 3 4 (5 6 7 8))
```

1.3 Pattern matching

Quoting, quasiquoting, unquoting and unquote splicing allow you to *build* abstract and complex nested lists. **Pattern matching** allows you to *use* those complex lists, that is, to perform some kind of case-analysis. Pattern matching in Racket is done using the `match` form[3].

```
(define (is-lt-1? num) ; function definition
  (match num
    [0 #t]
    [1 #t]
    [_ #f])) ; wildcard _ matches anything
(is-lt-1? 1) ;; => #t
```

The patterns `0, 1, _` appearing in the above example are all constants patterns in the sense that no variables appear in them. When patterns contain variables it is possible to extract subexpressions from the expression being analysed:

```
(match (list 1 2 3)
  [(list a b 3) b]) ;; => 2.
```

[2] https://docs.racket-lang.org/reference/quasiquote.html
[3] https://docs.racket-lang.org/reference/match.html

Using this pattern `(list a b 3)` we can match a list and its individual elements. The variables `a` and `b` appearing in the pattern are bound to the corresponding part of the expression we matched on (if the pattern matches of course).

Racket is also capable of pattern matching on quasi-patterns, which are quasiquote forms possibly containing variables. The previous example above can be reformulated as:

```
(match (list 1 2 3)
  [`( ,a ,b 3) b]) ;; => 2.
```

Quasi patterns can also contain ellipses to specify that some subexpression can appear an arbitrary number of times:

```
(define (2nd-elem alist)
  (match alist
    [`(,_ ,e ,rest ...) e])) ;;compare with `( ,_ ,e ,rest )  ?
```

Here `_` is bound to the first element which is discarded, `e` is bound to the second element, and `rest` is bound to a list constituted from the remaining elements. `rest` could have also been discarded as we are not using it in the returned expression.

Finally it is possible to guard patterns using `#:when` :

```
(match expr
  [`(,binop ,x ,rest ...)
   #:when (binop? binop) 'its-a-binop]
  [_ #f])
```

To enter a match guarded case, the pattern must match and the guarding condition must evaluate to true. This will be useful to write checkers for languages, analysing the syntax and raising informative exceptions when facing bad syntax.

A final example:

```
(define (prefix-to-infix expr)
  (define (binop? s)
    (member s '(+ - * /)))

  (match expr
    [`(,binop ,x ,rest ...)
     #:when (binop? binop)
     (if (empty? rest)
       `((,@(prefix-to-infix x)))
       `((,@(prefix-to-infix x))
         ,binop ,@(prefix-to-infix `(,binop ,@rest))))]
    [_ expr]))
```

Pattern matching is one of the most important concepts to grasp in this course.

1.4 Testing

With RackUnit[4] you will be able to write tests instead of "trying out" your racket code in the REPL. With tests you will gain confidence about the correctness of your solutions and it will make refactoring, updating, ... code more pleasant. Throughout this entire course we will work with RackUnit and it is required for the projects you will make as well.

Specifying a test submodule in your current file is trivial:

```
(module+ test
  (require rackunit) ;; rackunit required inside test submodule
```

[4] https://docs.racket-lang.org/rackunit/

```
)

(define (foo x) ...) ;; the code you would like to test

(module+ test
  (check-eq? (eval 1) 1)  ;; tests inside test submodule
  (check-eq? (eval '(+ 1 2)) 3)
  (check-eq? (eval '(+ 10 (- 5 2))) 13)
)
```

You can open the `test` submodule in multiple places in the file with `(module+ ..)` to add more tests. The tests can be run with the following command: `$ raco test session01.rkt` or you can just run your program in DrRacket.

We could also use a separate file to store our tests:

```
;; FILE: session01-test.rkt (as a convention use the
;;       suffix "-test" to indicate this is a test file)
(require rackunit ;; place this at the top of your file
         "./session01.rkt")

(check-eq? (eval 1) 1)
(check-eq? (eval '(+ 1 2)) 3)
(check-eq? (eval '(+ 10 (- 5 2))) 13)
```

2 Exercises

2.1 Hello World

- Define a function `(f name)` printing `"Hello <name>!"`. Don't forget the `#lang racket` line at the beginning of your file.
- Define a nested list `alist` with 3 levels of depth using the quote notation. For this kind of quick test you can use the `racket` command in a terminal. This will run a racket interactive interpreter which can be exited using `(exit)`.
- Write a list having as elements `alist, alist, alist`. Try to be as concise as possible.
- Define a list `blist = '(1 2 3 4)`. How can we then write an expression like `'(1 2 3 4 1 2 3 4)`?
- Define a function having lists of numbers of shape `blist = '(a b (c d ...) e)` as input and returning something like `'(a b (c+1 d+1 ...) e)`. Hint: look up for the `map` function in the Racket online documentation.

2.2 Pattern matching

1. **Invalid code.** What will be the exact result of evaluating this expression? How could we correct this code?

   ```
   (match '(1 2 (3 4) 5)
     ['(a b (c d) e) #t] )
   ```

2. **More ellipses.** What would this function do on `'( ["apple" 1] ["banana" 4] )` ?

   ```
   (define (foo expr)
     (match expr
       [ `(  [ ,left ,right ] ...  )   `( ,@left ,@right )  ] ))
   ```

More patterns can be found in the racket documentation.

2.3 A First Interpreter

Write a procedure `eval` interpreting the terms of the following language (bnf1) as numbers:

```
exp ::= int | (+ exp exp) | (* exp exp)
```

2.4 Stack Machine

We consider the language of a simple stack machine having `'plus`, `'times` and `'(push x)` (x is a number) as instructions. Define a procedure `exec` taking a list of instructions and a stack as inputs, executing the instructions 1 by 1 on the stack, and returning the top value of the final stack.

The following tests should pass:

```
(module+ test
  (check-eq? (exec '( (push 1) ) empty ) 1)
  (check-eq? (exec '(plus) '(1 2)) 3)
  (check-eq? (exec '((push 1) (push 2) plus) '()) 3)
  (check-eq? (exec
              '((push 1) (push 2) (push 3) plus times)
              '()) 5))
```

2.5 A First Checker (optional)

Here we simply define a valid stack language program to be a list of valid instructions. Define a checker `check-stack-lang` taking a stack language program as input. If all the instructions of the program are syntactically valid then the checker should just return the program. Else it should raise an error (using `(error "not a valid instruction")`). The following tests should pass:

```
(module+ test
  (define (check-stack-lang? instrs)
    (check-equal? (check-stack-lang instrs) instrs))

  (check-stack-lang? '(plus))
  (check-stack-lang? '(times))
  (check-stack-lang? '((push 1)))
  (check-stack-lang? '((push 1) (push 2) plus))
  (check-stack-lang? '(times plus times)))
```

2.6 Putting It All Together

Write a procedure `compile` that compiles bnf1 programs to stack machine language programs. The following tests should pass:

```
(module+ test
(module+ test
  (check-equal? (compile '1) '((push 1)))
  (check-equal? (compile '(+ 1 2)) '((push 1) (push 2) plus))
  (check-equal? (compile '(* 1 2)) '((push 1) (push 2) times))
  (check-equal? (compile '(+ (+ 1 2) (* 5 4)))
          '((push 1) (push 2) plus
            (push 5) (push 4) times plus)))
```

Remark. The `compile` procedure constitutes a quasi-trivial example of a compiler. It is composed of one single pass bnf1 $\rightarrow$ stack. The source language `bnf1` can be interpreted using the `eval` function we defined previously. Similarly the target language `stack` can be interpreted using the `exec` function. "Compiler correctness" could then be expressed as the fact that for each bnf1 program `p`, we have `(exec (compile p)) = (eval p)`.

```
(module+ test
  (define (check-eval-compile? expr expected)
    (let ([res (eval expr)])
      (check-equal? res expected)
      (check-equal? (exec (compile expr) '()) res)))

  (check-eval-compile? '1 1)
  (check-eval-compile? '(+ 1 2) 3)
  (check-eval-compile? '(+ (+ 1 2) (* 5 4)) 23)
  (check-eval-compile? '(* (+ 10 (+ 9 8)) (+ 10 (* 9 8))) 2214))
```