A crash course to Racket

Contents

1 Installation

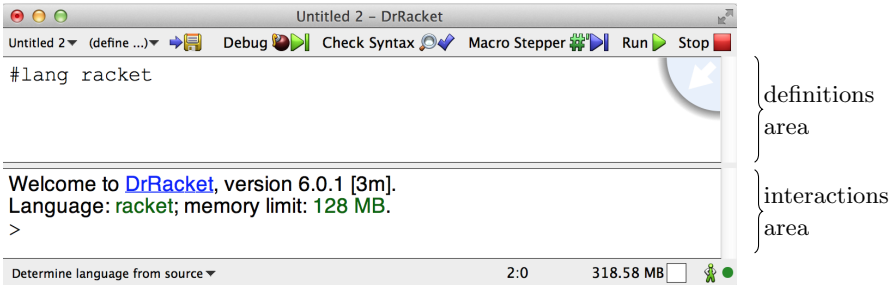
Racket is an interpreted language for functional programming. It is freely distributed for a large variety of platforms, including Windows, Unix, Linux and MacOS X, and has the same behavior on all platforms.

You can download and install Racket on your own computer from

`https://racket-lang.org`

2 Usage

The main way to use Racket is via DrRacket. DrRacket is the programming environment of the language. To start DrRacket, double-click its icon. A window made of two panels, as shown below, will open:



The definitions area

The top panel is the **definitions area**: It is the place where you usually edit and save your programs. A program consists of definitions. Most definitions in a program are function definitions. You can also include expressions to evaluate in a program.

Programs are run by clicking the Run button. As a result, the definitions will be loaded in the evaluation environment, and the printed forms of values of the expressions written in the program will be shown in the interactions area on separate lines.

The interactions area

This is the place where you interact with the interpreter of RACKET as follows:

1. You type in the expression you which to evaluate, immediately after the input prompt `>`
2. You click ENTER
3. The interpreter will **read** the expression, **eval**uate it, and **print** its value on the next line.

Afterwards, the interpreter waits for you to type in and evaluate another expression. This kind of interaction with the interpreter is called Read-Eval-Print loop (REPL for short).

3 The syntax of expressions in RACKET

RACKET is a descendant of Lisp. Lisp is the second oldest high-level programming language. It appeared in 1958, and was the favorite language of researchers working in Artificial Intelligence. RACKET inherits from Lisp a weird way of writing expressions, called **parenthesised prefix notation**. This means that all compound expressions are of the form

`(`$id\ arg_1\ \ldots\ arg_n$`)`

where

- id is the name of a function, special form, or macro
- $arg_1, \ldots, arg_n$ are subexpressions which represent the arguments expected by id.

For example, we write

- `(+ 1 2)` instead of `1+2`
- `(sqrt (+ (expt 3 2) (expt 4 2)))` instead of $\sqrt{3^2 + 4^2}$. In RACKET, `(expt m n)` computes $\texttt{m}^{\texttt{n}}$, and `(sqrt x)` computes $\sqrt{\texttt{x}}$.

The notation is a bit nasty for human readers (one has to type a lot of parentheses), but is easy to read and parse by the interpreter of RACKET.

You can ask RACKET to compute the values of these expressions in the interactions area. The Read-Eval-Print loop will produce the following results:

```
> (+ 1 2)
3
> (sqrt (+ (expt 3 2) (expt 4 2)))
5
```

3.1 Function calls

Most expressions are function calls. This means that *id* is the name of a function which expects n arguments.

The evaluation of a function call (*id* arg_1 ... arg_n) proceeds as follows:

1. First, we evaluate $arg_1, \ldots, arg_n$ and obtain the values $v_1, \ldots, v_n$.
2. Next, we call the function *id* to compute the value of (*id* v_1 ... v_n).

Remark: We can also have function calls of the form (e arg_1 ... arg_n) where e is an expression whose value is a function f. The evaluation of such a function call proceeds as follows:

1. First, we evaluate the $e, arg_1, \ldots, arg_n$ and obtain the values $f, v_1, \ldots, v_n$.
2. Next, we call the function f to compute the value of (f v_1 ... v_n).

For example, the evaluation of `(sqrt (+ (expt 3 2) (expt (3+1) 2))` can be depicted as follows:

`(sqrt (+ (expt 3 2) (expt (+ 3 1) 2)) ⇒ (sqrt (+ 9 (expt (+ 3 1) 2)))`
`⇒ (sqrt (+ 9 (expt 4 2))) ⇒ (sqrt (+ 9 16)) ⇒ (sqrt 26) ⇒ 5`

This way of evaluating function calls is called *strict* (or *call by value*) evaluation. Thus, RACKET is a strict functional programming language.

3.2 Special forms

Special forms are expressions (*id* arg_1 ... arg_n) where *id* is the identifier of a syntactic construct specific to the language of RACKET. The identifiers of special forms are predefined. Every special form has its own rules of evaluation. Often, special forms do not evaluate all their arguments, like function calls do.

Remark: To simplify the learning of RACKET, its implementers tried to keep the number of special forms as small as possible. Typical examples of identifiers of special forms are `and`, `or`, `if`, `let`, `define`, and `lambda`.
Below are the evaluation rules of some important special forms:

- (`and` arg_1 ... arg_n) computes the values v_i of arg_i starting from $i = 1$ up to n, and stops as soon as it computes a value v_i =`#f`. There are two possible outcomes:
  1. If all arg_i evaluate to $v_i \neq$`#f`, return the value v_n of arg_n.
  2. Otherwise, stop when finding v_i =`#f` and return `#f`.

  For example

```
> (and (+ 1 2) (+ 3 4) #t "abc" (/ 4 2))
2
> (and (+ 1 3) (< 2 1) 4/2 (sin 1.2))
#f
```

▶ `(cond (`$test_1$ $block_1$`)`
`...`
`(`$test_n$ $block_n$`))`

evaluates the expressions $test_1, \ldots, test_n$ in this order. There are two possible outcomes:

1. All values of $test_i$ are `#f`. In this case, the value of the `cond`-form is `#<void>`. The value `#<void>` has no printed form, and nothing is displayed as a result. However `#<void>` exists but we can not see it.
2. Find the first $test_i$ whose value is not `#t`. Then evaluate the content of $block_i$ and return the value of its last expression as the result of the `cond`-form.

Thus the `cond`-form behaves like `switch` instruction in `C++`. For example

```
> (cond ((> 1 2) "case 1")
        ((= 1 2) "case 2")
        ((< 1 2) "case 3"))
"case 3"
```

Remarks

1. There are two Boolean values in RACKET: `#t` and `#f`. Any value different from `#f` is called a *true* value. There are many true values (for example, `1`, `#t`, `"abc"`), but only one Boolean true value, which is `#t`.
2. RACKET allows to freely use the pair of parentheses `[ ]` instead of `( )`. For example, it is common practice to write `cond`-forms as follows:

   `(cond [`$test_1$ $block_1$`]`
   `...`
   `[`$test_n$ $block_n$`])`

3. Comments can be added to programs. A comment starts with the character '`;`' and extends to the end of line.

4 Writing programs

Several labworks are about writing programs. Programs are written in the Definitions area, and saved in text files with extension `.rkt` via the menu option

`File->Save Definitions` or `File->Save Definitions As...`

Typically, a program consists of variable definitions of the form `(define` *id expr*`)`. A variable definition is a special form whose evaluation has the following effect:

1. The interpreter computes the value v of *expr*.

2. The interpreter assigns name id to v. From now on, we can use id to refer to v.

For example:

`> (define x (sqrt 10.)) ;` give name `x to the numeric value of` $\sqrt{10}$.

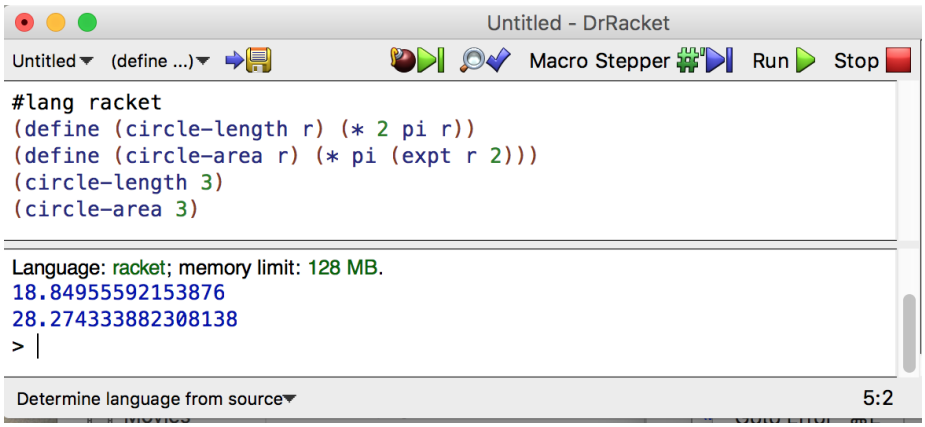
Most often, programs contain definitions of functions. Function names are variables. The definition of a function f which, for input arguments $x_1, \ldots, x_n$ evaluates the content of *block* and returns its value, is written as follows:

`(define` f `(lambda (`x_1 `...` x_n`)` *block*`))`

or in the simplified but equivalent form

`(define (`f x_1 `...` x_n`)` *block*`)`

For example, the following program defines functions to compute the length and area of a circle with radius `r`, and computes the length and area of a circle with radius `3`. To run it, click the Run button:



5 List of useful predefined functions

RACKET has several predefined functions that we can freely use when writing programs. Below is a concise list of the most important predefined functions.

5.1 Numeric functions

function	arguments	return value
`+`	0 or more	sum of arguments
`-`	1 or more	difference of arguments in let to right order
`*`	0 or more	product of arguments
`/`	1 or more	quotient of arguments in left to right order
`max`	1 or more	maximum of arguments
`min`	1 or more	minimum of arguments
`truncate`	*num*	integer part of *num* (digits to the left of the decimal point)
`sqrt`	*num*	square root of *num*, $\sqrt{num}$
`abs`	*num*	absolute value of *num*, $\|num\|$
`expt`	*num pow*	exponentiation (*num* raised to *pow*), num^{pow}
`quotient`	num_1 num_2	quotient of num_1 divided by num_2
`remainder`	num_1 num_2	remainder of num_1 divided by num_2

The following examples illustrate the behaviour of some of these functions when they are called with an unusual number of arguments:

```
> (+)          > (*)          > (/ 4) ; in general, (/ n) computes 1/n
0              1              1/4

> (/ 2 3 4) ; division is left-associative and computes (2/3)/4
1/6
```

Rational numbers are always reduced to lowest terms. For example, the evaluation of `(/ 2 3 4)` produces the value 1/6 instead of 2/12.

5.2 Functions on lists

Remember that the printed form of a list value is

'(w_1 ... w_n)

and that it can also be used as an input form. The most important functions on lists are:

- `(list? l)` is the recognizer of lists. It returns `#t` if `l` is a list, and `#f` otherwise.
- `(null? l)` is the recognizer of empty. It returns `#t` if `l` is `null`, and `#f` otherwise.
- `(length l)` returns the length, that is, number of elements of list `l`.
- `(cons v l)` constructs a new list by adding value `v` in fromt of list `l`.
- `(car l)` returns the first element of list `l`. It fails if `l` is empty list.

- `(cdr l)` returns list `l` without first element. It fails if `l` is empty list.
- `(reverse l)` constructs a new list consisting of the elements of list `l` in reverse order.
- `(append` $\mathtt{l}_1 \ \ldots \mathtt{l}_n$`)` appends lists $\mathtt{l}_1, \ldots, \mathtt{l}_n$.
- `(list-ref l i)` returns the `i`-th element of list `l`. List elements are indexed starting from 0, thus `(list-ref l 0)` returns the first element of `l`. Note that `(list-ref l 0)` is equivalent with `(car l)`.

The following predefined higher-order functions on lists are also very useful:

- `(map f l)` computes the list

  `(list (f` v_1`) ... (f` v_n`))` if `l` is `(list` v_1 ... v_n`)`
- `(filter pred l)` computes the sublist (`list` $\mathtt{v}_1 \ \ldots \ \mathtt{v}_p$) of elements of list `l` for which (`pred` $\mathtt{v}_i$) holds.
- `(foldl f` $\mathtt{v}_0$ `l)` computes the value of (`f` $\mathtt{v}_n \ \ldots (\mathtt{f} \ \mathtt{v}_1 \ \mathtt{v}_0) \cdots$) if `l` is the list (`list` $\mathtt{v}_1 \ \ldots \mathtt{v}_n$).
- `(foldr f` $\mathtt{v}_0$ `l)` computes the value of (`f` $\mathtt{v}_1$ (`f` ... (`f` $\mathtt{v}_n$ $\mathtt{v}_0$) ...)) if `l` is the list (`list` $\mathtt{v}_1 \ \ldots \mathtt{v}_n$).
- `(apply f l)` computes the value of `(f` v_1 ... v_n`)` if `l` is the list `(list` v_1 ... v_n`)`.
  The first argument of `apply` must be a function that can take any number of arguments. Such functions are called **variadic functions**. Examples of variadic functions are + and *.