

MAU11601

Lecture 00

2026-09-22

Introduction

MAU11601 Introduction to Programming

John Stalker stalker@maths.tcd.ie

<http://www.maths.tcd.ie/~stalker/11601/>

This module is meant to teach programming in general, not any particular programming language, but we have to pick a language and I've picked Racket, for a few reasons:

- Easy installation on (almost) every system.
- Good documentation
- Good textbooks
- Simple user interface
- Fairly logical design

I'll talk about various language design choices as they come up.

Resources

Slides and other materials will appear on the module website.

Nothing will appear on Blackboard. Do not look for anything there. Do not attempt to use it to communicate with me.

The home page for the Racket project is <https://racket-lang.org/>. There you can find download links and documentation.

The textbook for the module is How to Design Programs, Second Edition by Felleisen, Findler, Flatt and Krishnamurthi. It's available for free online at <https://htdp.org/>. It will be a few weeks before we start using it, but when we do you are responsible for reading it.

Racket is a kind of Scheme. There are other good introductions to Scheme, like <https://files.spritley.institute/papers/scheme-primer.html> from the Spritley Institute or Abelson, Sussman and Sussman's Structure and Interpretation of Computer Programs <https://sarabander.github.io/sicp/html/>, which may be helpful.

Assessment

- There will be an exam worth 70% of your mark. I will give you more information about its contents and structure later.
- There will be weekly assignments worth 0% of your mark. Solutions will be posted. They're meant to aid your understanding.
- There will be a group project worth 30% of your mark, due the end of the term. The group project is a much better simulation of the sort of programming that will be useful to you in the future. It's at a larger scale, which means you need to think carefully about how to organise it and you need to document it properly. The program you will need to develop is an interactive one, rather than the batch type programs more typical of introductions to programming. It's also a collaborative activity. The project is to design, write, and document, a game. More details are on the website and still more will be provided in a couple of weeks.

Racket

Racket is a (mostly) functional, statically scoped, garbage collected, dynamically typed language with eager evaluation.

Most of those words won't mean much to you now but I will explain them later.

As I said earlier, it's available for (free) download from the website. It should be easy to install on Windows, OSX, Linux, *BSD, etc.

It comes with its own graphical user interface, called DrRacket. You don't have to use the GUI but it has a lot of convenient features so I'll always use it for demonstrations. It has two windows, the definitions window and the interactions window. You mostly use the first to write programs and the second to test them.

There are lots of sublanguages. Initially we'll use the Beginning Student Language and then the Intermediate Student Language. They're meant to be easier, but less powerful, than the full language.

To use BSL, put `#lang htdp/bsl` at the top of your programmes.

You can also select it from a menu at the bottom. Don't!

It's time for a demo!

Expressions

Programs are built from expressions. The semantics of programs is determined by how expressions are evaluated.

There are three main ways of writing arithmetic expressions:

- Infix, i.e. the one you learned in school, like $1 + 2$
- Prefix, a.k.a. Polish notation, like $+ 1 2$
- Postfix, a.k.a. reverse Polish notation, like $1 2 +$

Infix notation requires associativity and precedence rules, and parentheses to override those rules as needed, to distinguish $1 - 2 - 3$ from $1 - (2 - 3)$ and $1 + 2 * 3$ from $(1 + 2) * 3$.

Prefix and postfix don't need parentheses. $- - 1 2 3$ and $- 1 - 2 3$ are distinct, as are $+ 1 * 2 3$ and $* + 1 2 3$.

Racket is a Scheme and Scheme is a Lisp. Lisps are prefix languages and so shouldn't need parentheses, but force you to use them anyway.

Infix and parentheses

Parentheses are the least popular part of Lisp, so why are they there?

- They allow functions with variable numbers of arguments, e.g. `(+ 1 2 3)`.
- They make it easy to identify subexpressions. With infix, `2 * 3` is a subexpression of `1 + 2 * 3` but `1 + 2` is not. This requires the rules of precedence though. With fully parenthesised prefix it's clear why `(* 2 3)` is a subexpression of `(+ 1 (* 2 3))` while `(+ 1 2)` is not. Subexpressions are substrings with properly matched parentheses.

You may find fully parenthesised prefix notation annoying until you get used to it though. Here are some ways to cope with it:

- Use the facilities of DrRacket (or your preferred editor)!
- Racket actually allows you to put the operator anywhere, if you put dots before and after, like `(1 . + . (2 . * . 3))`.
- Racket is designed to have extensible syntax. You can create a module which parses expressions, applying the standard associativity and precedence rules, so that `{1 + 2 * 3}` produces 7, as expected.

In fact, I did. It doesn't work with BSL but it does with ISL and Racket.