Essentials of Programming Languages

https://proglang.informatik.uni-freiburg.de/teaching/konzepte/2018ss/

Language 1 – Introduction and Arithmetic expressions

2018-04-18

The exercises for this lecture will be composed of a succession of language definitions, each with potential extensions. Your task is to implement these language definitions with *PLT Redex*, a Racket framework for implementing language semantics. Knowledge of Racket (or any other Lisp dialect) is not required. The documentation is available at the URL below. You are encouraged to consult the tutorial and the reference manual.

https://docs.racket-lang.org/redex/index.html

To get started, install the *racket* package on your system and launch the IDE "Dr Racket" (`drracket` on the command line). That's it.

Arithmetic expressions

The language of arithmetic expressions correspond to the following grammar:

$$\langle val \rangle \ ::= n$$

$$\begin{aligned}\langle exp \rangle \ ::= & \ \langle val \rangle \\ & \mid (+ \ \langle exp \rangle \ \langle exp \rangle) \mid \ (- \ \langle exp \rangle \ \langle exp \rangle) \mid (* \ \langle exp \rangle \ \langle exp \rangle)\end{aligned}$$

A base implementation is provided in the `arith.rkt` file available on the course's page. This base implementation only handles the + operator.

Exercise 1 (Arithmetic expressions)
Extend the language with − and ∗. Consider the use of metafunctions (defined using `define-metafunction`) to factorize operator evaluation.

Note: We do not implement the division operator / for now. Indeed, this requires that the semantics can fail. We will see later on how to formalize this.

Exercise 2 (Boolean expressions)
Extend the `arith` language with boolean expressions and tests using the `define-extended-language` function.

$$\begin{aligned}\langle val \rangle \ ::= & \ \ldots \\ & \mid \texttt{true} \mid \texttt{false}\end{aligned}$$

$$\begin{aligned}\langle exp \rangle \ ::= & \ \ldots \\ & \mid (< \ \langle exp \rangle \ \langle exp \rangle) \mid (\wedge \ \langle exp \rangle \ \langle exp \rangle) \mid (\vee \ \langle exp \rangle \ \langle exp \rangle) \mid (\neg \ \langle exp \rangle) \\ & \mid (\texttt{if} \ \langle exp \rangle \ \langle exp \rangle \ \langle exp \rangle)\end{aligned}$$

Write new tests that exercise this language.

Exercise 3 (Big step semantics)
Provide a big step semantics for the language of boolean and arithmetic expressions. You can use the `define-judgment-form` function. Below is a judgement `==>` that takes an expressions `e` in input mode (`I`) and a value `v` in output mode (`O`).

```
(define-judgment-form arith
  #:mode (==> I O)
  #:contract (==> e v)

  ;; Write rules here

  )
```

Exercise 4 (Bonus: Equivalence of semantics)
Use the random testing facilities to test that the big and small step semantics are equivalent.

Exercise 5 (Bonus: N-ary operators)
Extend the boolean and arithmetic operators to operate on any numbers of arguments.