Wrench:

Named arguments for Racket

By Nitan Singh

Summary

This branch of racket extends the language and implements name-based argument definition. This is similar to how languages like python and Swift handle arguments, where arguments have a name and have to be called with those names in the call arguments. For example:
Def foo (x, y), and the call would be foo(x: 10, y: 20). I kept the typing in the definitions generic if I can as to not lose dynamic typing. The extension is called Wrench, because of the versatility of the method call syntax. The submission contains 2 folders, Wrench-Iniquity and Wrench-Loot. Both versions extend their own respective languages, and although their compiler-level implementations are vastly different, the syntax and results are the same.

Implementation

The syntax is as follows: function definitions remain the same (define (foo x) (…body…), however, argument calls can either be positional as usual (foo (sub1 10)), or named (foo (: x (sub1 10)). The syntax for name-based arguments is as follows: the operator (:), followed by the parameter symbol (x), followed by the assigned value (sub1 10). Wrench can detect whether named or positional calling is used based on syntax.

My implementation essentially assimilates named arguments into the existing positional structure, hence the “rearrangement” focus of the implementation. This also allows it to keep positional calling, the only thing extra required is syntax detection.

The high level implementation of both the Loot and Iniquity versions is the same; if the call arguments don’t contain the : operator (with the correct following values), then it will continue with positional calling as usual. However, if named argument syntax is detected, it will retrieve the parameter list from the appropriate function, rearrange the arguments so the arg variables match the parameters, retrieve their values as if they were positional, and call the function. However, lower level implementation in the compiler is different for both.

For Iniquity, the rearrangement happens before any assembly instruction. It is done solely in Racket, and then normal compile and call instructions occur. This has the disadvantage of the functions are not stored, hence why recursive calls are unstable. Additionally, the Iniquity interpreter runs on a similar framework, allowing the interpreter to be implemented as well.

In Loot, the function parameters are stored in the stack environment, and when the call is compiled, the parameters are retrieved from the stack and the arguments are then rearranged. Although this is much more complicated, the finished result allows for recursion and storage of the function. Syntax detection also allows for mixed call types between lines. For example, a program could contain a call in named syntax, and another call in positional syntax.

Testing

Both versions contain test files that encompass the features of their implementations. For Iniquity, recursive calling is unstable, but otherwise all vanilla features remain intact. For Loot, recursion and all other vanilla features are intact, however named arguments only extend to user defined functions. Lambda named-arguments are currently unstable.

Current bugs

- Lambdas can be unstable when using nested calls
- Large nested functions use slightly more memory than necassary, due to excess memory needing to be allocated for stack alignment