The programming language: Hyra

DaOnlyOwner

March 28, 2018

1 Design Philosophy

Hyra is a programming language that was born because I love C++ ($\geq$ 11) and Python (and Rust). C++ for being fast while still maintaining some sort of comfort, at least with newer versions of C++. And Python for its simplicity and syntactic beauty. Don't even get me started with the standard python library. Python is runnable pseudocode, it's that good. So why a new language? The answer is: I probably just want to design a new one. And the second answer: There are problems with both languages I mentioned. Let's see:

- C++ lacks clarity
- C++ lacks a clear vision
- C++ lacks good metaprogramming support
- C++ is complicated
- Python is slow
- Python is managed, thus not feasible for low level programming

Rust would be the perfect candidate for my new fav language. But it gets in your way with trying to make everything safe without a GC. In my opinion this is horrible and leads to complicated source code. So if Python and C++ had a child which inherits the looks of Python, but the inner workings of C++, with proper metaprogramming capabilities too, I would freak out.

2 Compiler Implementation

2.1 Features

2.1.1 Boolean Expression chains

In maths you can compare values like this: $2 < 4 < x > 3 == 2 < 4 \&\& 4 < x \&\& x > 3$. In Hyra you can do that, too!

2.1.2 Functions

A function can be called with either f(arg1,arg2,arg3) or arg1,arg2,arg3@f; Looks nicer and fewer things to write. At compiletime: arg1,arg2,arg3@#f; Macros !arg1,arg2,arg3@f ! marks the start of a macro

2.2 Syntactic Analysis

2.2.1 Distinctions

I am not using a Lexer that parses the regular part of my languages and spits out tokens. I tried out Flex and even wrote my own generator, but I found them not to be useful and very ugly to look at. It's just an preprocessing step which can be easily handled inside the grammar itself. Maybe this is slower or maybe not. If I find the time and bootstrap the compiler I can imagine implementing something like a Lexer.

2.3 Operator Precedence Parsing

Note that I refer to "Precedence" as "Binding". So + has a lower precedence than * because its binding is weaker; Assume we have already parsed a binary expression whose operator op1 has a precedence of p1. So we have a tree Op1 which has a left and a right child. Assume per induction that those children are already parsed correctly and according to the precedence rules. Now we come across another binary operator op2 with precedence p2. The situation looks like this: $A\ op1\ B\ op2\ C$. If $p1 <= p2$ (e.g. $+ < *$) then we construct a new tree with Op2 being the father and Op1 his left, C his right child. The expression looks like this now: $((A\ op1\ B)\ op2\ C)$ If however $p1 > p2$ (e.g. $* > +$), then we have to rearrange the tree. In fact we have to steal the right child of Op1 and make it the left child of Op2. C becomes the right leaf of Op2. Then we connect Op1 and Op2 using Op2 as the right tree of Op1 and A as the left tree resulting in something like this: $(A\ op1\ (B\ op2\ C))$. Unary Operators are easy to parse, even if they are ambiguous like -5 or $++x$. If we come across an operator and we currently don't have any valid expression tree because the parser expectss another operand then it's unary. This concept can be generalized and applied to different operator types.