Building a Statically Typed Interpreted Language

Jake Armendariz

Introduction

As the world of computer science expands the number and complexity of programming languages are increasing rapidly. Production level languages often grow in complexity over time leading to a bulky design in which simplicity is traded off for semantic desires. This quarter, my goal was to build a small language that is as simple, clean and structured.

Design

In order to direct my design, I tried to build an ideal language for a beginner. Often, when learning to program, the first question a beginner asks is "Which language should I learn?". The most common languages I hear in response are Python and Java, both of which fall short in a couple of categories. Python is missing types and lacks a much-enforced structure. Java on the other hand, requires far too much overhead and can be intiminating to a new programmer.

My goal was to build a language that fit in the middle of these two, simple like Python, but with a structure similar to Java/C. I decided to call this language Worm, named for its small size, and my appreciation for Python's simplicity.

A new programmer should learn about types, functions, arrays, hashmaps. In order to express these types explicitly, types are expressed in the format `vartype varname` or `int x`, similar to Java or C. Functions must have explicit return types and parameters, and arrays and maps are built-in types to reduce syntactic complexity when learning new data structures.

The "Hello World" of Worm is decidedly simple.

```
fn main() -> int {
    print("Hello, World");
    return 0;
}
```

Types

```
int, float, char, string, map,
int[], float[], char[], string[]
```

Along with some basic rules, I added some custom, unique semantic features to Worm that I thought would help enforce good practice in a new programmer.

Pass by Value: All parameters are pass by value, not pass by reference.

No Void Function Because all functions are pass by value, every function must have a return value, otherwise, they would have no purpose.

No Declaration, only Assignment In order to reduce the number of errors, every variable must be assigned a value when it is declared

Array Definition I tried to create a unique way to declare arrays in Worm. Every array can be defined either by a function return or by a `[value; size]` so if a programmer wants an array of all zeros they can do `[0; 20]`. For more dynamic changes, I allow piping of values, so if `[|i| i; 20]` builds an array counting 0-19. Or even `[|index| index^2; 100]` builds an array of size 100

map type Integrated into the language is a custom hashmap type that can map from any variable type into any other. Right now there is no type of checking involved in this stage, which is clearly a large problem. So to solve this problem, although the map type is from any type to any other, once a key is assigned to a value, the variable type initially assigned to the key is stuck for the lifetime of the hashmap.

Thus, if someone does `map["id"] = 0` then `map["id"] = "0"` there will be a type error, but `map["id"] = 2` causes no such violation

Static Analyzer and Error Checking

I have found good error messaging to be an essential part in a good programming language, so I tried to spend a lot of time on the type system and proper error messages. I used a parser library to build the Worm interpreter, and it limited my ability to create good error messages greatly, as I couldn't reference specific line numbers or text when writing error messages. Instead, I used function names, and function lines to indicate where the bug occurred.

Static analysis is performed after parsing and before execution, it will analyze the AST by checking every line to ensure that types match by inspecting each expression's values/variables/function calls all contain the same type. It will also look at possible errors in variable scope, existence, function returns, parameters, and once finished it will print all of the errors, count them, if the number of errors > 0, then the program will stop before reaching the evaluation stage.

Sample Program

In order to demonstrate Worm's capabilities as a working language, I wrote some sample programs in it. This is part of the working program that can parse and evaluate an expression string. I wanted to leave this as a demonstration of Worm.

```
import "worm/wormstd.c";

/* extracts a substring that represents an integer */
fn int_substring(string s, int index) -> string {
    string result = "";
    int i = index;
    while index < len(s) & s[index] != ' ') {
        result = result + s[index];
        index = index + 1;
    }
    return result;
}

/* Convert the postfix notation to a tree */
fn postfix_to_tree(string postfix) -> map {
    map[] stack = [{}; 0];
    int i = 0;
    while i < len(postfix) {
        char curr = postfix[i];
        if curr == ' ' {
            skip;
        } else if is_operator(curr) != 0 | postfix[i+1] != ' ' {
            /* operand */
            map node = {};
            node["is_op"] = -1;
            string s = int_substring(postfix, i); /* retrieves the substrin
            node["data"] = parse_int(s); /* parses the integer string, gets
            i = i + len(s) - 1;
            stack = push_map(stack, node);
        } else {
            /* is an operator */
            map node = build_node(curr);
            node["right"] = stack[len(stack) -1];
            stack = pop_map(stack);
            node["left"] = stack[len(stack) -1];
            stack = pop_map(stack);
            stack = push_map(stack, node);
        }
        i = i + 1;
    }
    return stack[len(stack)-1];
}

/* executes the tree */
fn evaluate_tree(map root) -> int {
    if root["is_op"] == 0 {
        int left = evaluate_tree(root["left"]);
        int right = evaluate_tree(root["right"]);
        if root["data"] == '+' {
            return left + right;
        } else if root["data"] == '-' {
            return left - right;
        } else if root["data"] == '*' {
            return left * right;
        } else if root["data"] == '/' {
            return left / right;
        } else {
            print("unknown operation");
            return -1;
        }
    } else {
        return root["data"];
    }
    return 0;
}

/* turns string into postfix, then a tree, then it evaluates tree with a
pres order traversal */
fn evaluate_expression(string infix) -> int {
    infix = pop_str(infix); /* remove the \n */
    string postfix = infix_to_postfix(infix);
    map root = postfix_to_tree(postfix + ' ');
    return evaluate_tree(root);
}

fn main() -> int {
    string infix = user_input();
    return evaluate_expression(infix);
}
```

Conclusion

Building Worm is not very special nor relevant to modern research in programming languages. But I have found a lot of value in how I understand language design as well as a new perspective while programming in general. I wrote this project in Rust, and I gained a much deeper sense of appreciation for the Rust language design decisions, error handling (Rust result object) as well as the complexities of the compiler as I learned more about it.

Although I spent many hours on Worm, I hope to work on it more this summer. I plan on making my static analysis work on state internal methods so that I can edit the AST before running, rather than the current method of passing copies/clones of the state. This would help add some sugaring into the language that wouldn't be needed in the executable. This project has also sparked an interest in compilers/LLVM and I would really like to get worm compiling to LLVM intermediate representation. There are a few Rust bindings for LLVM that take time to get working correctly, but it's very intriguing. I've had a lot of fun on this project, and I hope its the first of many small languages that I build for fun.

Sources

I tried referring to research and articles on programming languages and type systems, but nothing was very useful for what I was working on. So my only references are the language and library references I used.

- **Parsing**: https://pest.rs/book/
- **Programming**: https://doc.rust-lang.org/book/