Physics Notes

Author

October 23, 2022

Contents

1 Analysis

1.1 Translator Design

A translator is a piece of utility software that takes one set ofprogram source code and turns it (translates) into the program sourcecode of another language. They are incredibly versatile pieces ofsoftware, and as such there are many different ways of desinging andimplementing translators. Generally, they come in three different forms.Most basic are assemblers, which take assembly language code andtranslate it to executable machine code. Then, we have interpreters andcompilers, which translate high level languages. These are significantlymore complicated, since they also have to take into account the codesemantics and structure. Interpreters and compilers differ generally intheir output. And interpreter translates line by line (or construct byconstruct) and executes as it goes, stopping when it reaches a haltingcondition; this could be an error or just the end of the code. Compilers translate the entire source code and output an executable machine code file. It always stops when it reaches the end, forming a list of errors as it goes.

Although interpreters and compilers have separate outputs, they follow very similar steps. The way that these steps are connected, so the path that the data takes, is part of what gives each language and translator its individual flavor.

1.2 Proposed Solution

My proposed solution is called `Lilac`. It is a minimal functional programming language inspired by the syntax of Pinecone and Boa, along with the function logic of Haskell. Here is an example source file:

```
main: ( print "Hello World" )

addToThree: ( fn x -> 3 + x )

fooBarBaz:
  let (
    byThree: n % 3,
    byFive: n % 5 )
  in (
    fn n -> byThree ? "Foo" | byFive ? "Bar" | "Baz" )
```

1.3 Objectives

By the end of this project I aim to have written an interpreter for the Lilac programming language which can: - Execute arithmetic - Execute boolean arithmetic and comparison - Define and store variables - Define and use functions - Conditional branching statements - Recursive function calls and definitions

In addition to this, the interpreter should provide a REPL and the ability to run source files. It should also be able to import them into the REPL or another script.

2 Design

2.1 Stage One: Framework and Interface

2.2 Stage Two: Lexical Analysis

2.3 Stage Three: Parsing

2.4 Stage Four: Execution

Execution model: The Tree Machine The parser, thanks to my design of Lilac, outputs a recursive binary parse tree. At this point there are a few ways to proceed with execution. I considered doing a reverse order traversal, and then using a simple stack/virtual machine. However, it is difficult to implement code branching and lay evaluation this way. This is due to the fact that it is a bottom up technique, so at the leaves it is completely unaware of the context above it. Instead, i decide to manipulate the AST directly, and to execute it recursively in a top down manner. This is handled by my driving class, the `TreeMachine`. It is defined as follows:

```
TreeMachine:
    env_monad :: EnvMonad
    tree      :: Tree

    execute   :: Tree -> (*output)
```

The TreeMachine is the object that is responsible for running the code. In the REPL context it will only do the execute function on one line, but in a script context it will also handle imports, (running the machine on another source file while ignoring main), and running `main`.

Monads: Stack A central issue in my design is deciding where behaviours go, and who handles what. To solve this, I decide to opt for a monadic model. A monad is an object that contains some value, and handles behaviors and side effects relating to that value. This is helpful for separating behavior from types, and to improve the simplicity of the code. For example, the `Maybe` monad allows for safe computation. The data stored within it can be either `Just x` or `Nothing`. If a computation fails, the monad catches this and makes the value `Nothing` - our computation is saved. The two main components of a monad are the `return` function and thhe `bind` function. Borrowing the type signatures from haskell, these look like:

```
return :: Monad m => a -> m a
bind (>>) :: Monad m => m a -> (a -> m b) -> m b
```

Here we see that `return` takes a value of type `a` and returns a monad containing a value of type a. In an object-oriented way, this is the class constructor. We also see that bind, which also has the operator >> takes a monad of type `a` and a function that returns a monad of type `b`, then returns a monad of type `b`. Simply, it takes a monad, applies the function to the value inside the monad, then returns the output from that. This is the central benefit of the monad; the behavior is completely separate from the value.

For my call-stack, which is used to evaluate expressions, I use this model. I create three classes: `Stack`, `Node`, and `StackMonad`. The stack is defined in a dynamic linked way using the `Node` class in the following way

```
class Node:
    value :: Token
    next :: Node

class Stack:
    top :: Node

    pop :: Stack -> Stack
    push :: Stack -> Token -> Stack
```

Then, I define the `StackMonad` like so:

```
class StackMonad
    stack :: Stack
    out :: List[Tokens]

    (>>) :: StackMonad -> function -> StackMonad
```

The functions that the bind will take are the stack operations, and bind will simply implement them. The output of popping the stack is stored in the `out` list of the monad.

Monad: Environment and Actions I have an `Environment` class, which can also be thought of as a scope. This class looks as follows:

```
class Environment:
    stack_monad :: StackMonad
    table :: Dictionary
    tree :: Tree (optional)
```

We can see here that the environment holds only data, and no behaviors. It is contained within my `EnvMonad` class:

```
class EnvMonad:
    env :: Environment
    trace :: List

    bind :: EnvMonad -> Action -> EnvMonad
    consume :: EnvMonad -> Environment -> EnvMonad
```

The `bind` function is a feature of monads. It takes an `EnvMonad` (i.e. self) and an `Action`, runs the action on the data of the monad, then returns the result wrapped in a new monad. The `Action` is a function wrapped in a class:

```
class Action:
    run :: Environment -> Environment
    check :: (*args) -> (*outputs)
```

Each component of the language gets its own action, and the interface for each action is strictly the same. This allows me to easily extend the language, simply by adding new actions

3 Implementation

4 Testing

5 Evaluation