Implementation of a Functional Programming Language

Jason Koenig
Computer Systems Lab 2009-2010

Abstract

Scripting languages have increased greatly in popularity in recent years with the growing power of computers. The trade off of runtime and programmer time is increasing favoring using more runtime. However, most current scripting languages are imperative. A language is developed which is primarily functional in style. The language has novel features which allow the base interpreter to be small in size, will the lack of features such as `eval` allow the programs to be optimized easily.

Introduction

The purpose of my project is to develop a functional style programming language. The language is similar to Lisp, but contains features to make it friendlier to imperative programmers. The initial version will be interpreted, but I expect to eventually at least partially compile code.

One goal is to make the interpreter as small as possible, allowing the language to easily be embedded in other programs. This will allow my language to be used both on its own, and embedded as a scripting language like Python.

Beyond the implementation, I will also develop a series of tutorials and example programs that will assist in learning my new language. This will be important if my language is to become anything other than a toy language.

Sample Program

```
let
   X = 5,
   Y = 3+X,
   Z= {t| t+2}
in
   Z.(X+Y) # output is 15
```

Design

In my language, like other functional languages, a program is executed by evaluating the main expression. This expression is usually composed of sub-expressions, which are then composed of sub-expressions, and so on.

Several novel ideas were incorporated, such as using an explicit character for function application (.) rather than whitespace. This simplifies the parser greatly, which was a design goal.

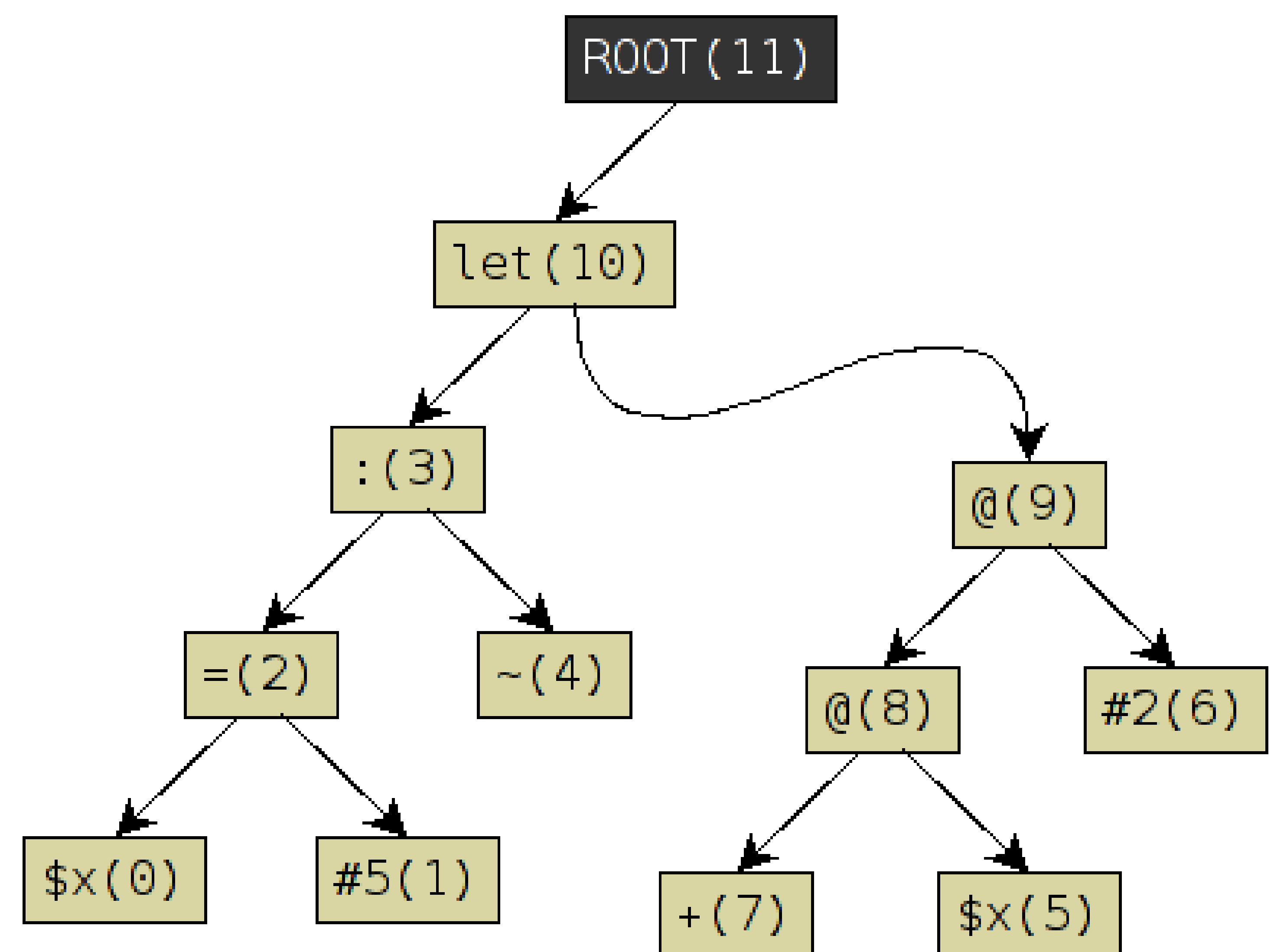

Figure 1. The graph after the parser stage. Notice that the let expression holds a list of assignments (in this case only one). The numbers in parentheses are the node numbers, which are like pointers to the node.

Implementation

The interpreter is divided into a number of relatively independent sections. The first part is the lexical analyzer, which turns the sequence of characters into a sequence of tokens. The parser turns the linear token stream into the first revision of the graph. This output is visualized in Figure 1. The parser hands it's graph to the optimizer. The optimizer is responsible for the transformation of the graph from a lexical one into one which the executor can use. This involves the removal of variable names, and the reduction of constructs such as let. The executor is the final stage of program execution. The interpreter is responsible for walking the graph and performing the instructions found there. The output of the program is produced at this step.

Results

The language currently can perform reduction of complex mathematical expressions. It also has support for a large number of the final operators that are part of the core language. It can perform let reduction, as well as execute simple user defined functions. There is no support yet for imperative programming. Recursive functions do not work properly, but they should with the implementation of lambda lifting in the optimizer (support already exists in the executor). Both a program that reads in source code from a file, as well as one that accepts user input from the command line were created.