Functional Programming Language Design and Implementation
TJHSST Computer Systems Lab 2009-2010

Jason Koenig
Latimer Prd. 5

June 15, 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.

1 Introduction

The purpose of my project is to develop a functional style programming language. This include both the language definition and a sample implementation. 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. The first interpreter has been written in Python, but the final implementation will be in C for speed. I also plan to include some optimization so that the language is not too slow. This can be everything from simple things such as constant propagation to complex things such as an optimizing G-code system.

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. Another goal is to create a language that allows both functional and imperative styles in the same language. Some of these features are similar to Lisp and Javascript, such as a definite execution order and allowance of local variables. I will also implement control structures such as while and `foreach`.

2 Background

There have been numerous functional languages over the years. The heaviest influences are from Lisp. Lisp has a definite execution order, and has support for assignable local variables. Lisp is very complex, however, and the interpreter is very large. It also has a complex and diversified family of languages, which makes it quite difficult to learn 'Lisp', rather than Scheme or Common Lisp or one of it's derivatives. Further, Lisp has a very large standard library, which makes it difficult to port cleanly. My language would be focused on simplicity and speed, rather than on supporting every possible feature. This makes it easy to port.

Haskell shares more of a syntactic representation with my language. It however, is completely functional, which means no variable assignments. It is also lazy, which means computation is deferred until the last possible moment. Thus things like function side effects are not allowed. In languages like C++, sometimes expressions are evaluated simply for their side effect, like accessing a memory location to bring it into the cache. In Haskell, this is impossible, as simply accessing something is not enough to force its evaluation. This in turn forces the programmer into the functional style, which makes some operations, like input and output, harder. It also requires a shift in thought process to understand. I want to avoid this as much as possible in my language. By supporting imperative programming, I will ease more people into the functional style, and give my language a higher chance of success.

Other similar languages focus on having a strong mathematical foundation. My language is not designed with mathematical elegance in mind, but rather with being a concrete language that is useful. Lisp uses the same representation to represent code and data. While this makes certain kinds of programming easier, it is much more difficult to optimize, because the original representation must be retained and the optimized version must be recreated when the code is modified.

Further, these languages tend to provide functions such as `eval` that allow a string to be executed as if it were code. While this is useful in producing flexible code, in practice its uses are extremely limited and can be avoided by proper software engineering.

3 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.

A program in this language is encoded using ASCII formatted text, which may be in a file, on a stream, in a buffer in memory, etc. The program is first divided into tokens. Then a tree is created from these tokens, which is then executed. The language is sometimes ambiguous. In these cases, parentheses are required to disambiguate.

This language uses infix notation for most expressions. The exception to this rule is the control structures, which are denoted with a special initial token and possibly a series of internal tokens. Thus the type of a given subexpression can be determined solely by it's first token. Thus the language can be parsed by a simple recursive descent with backtracking

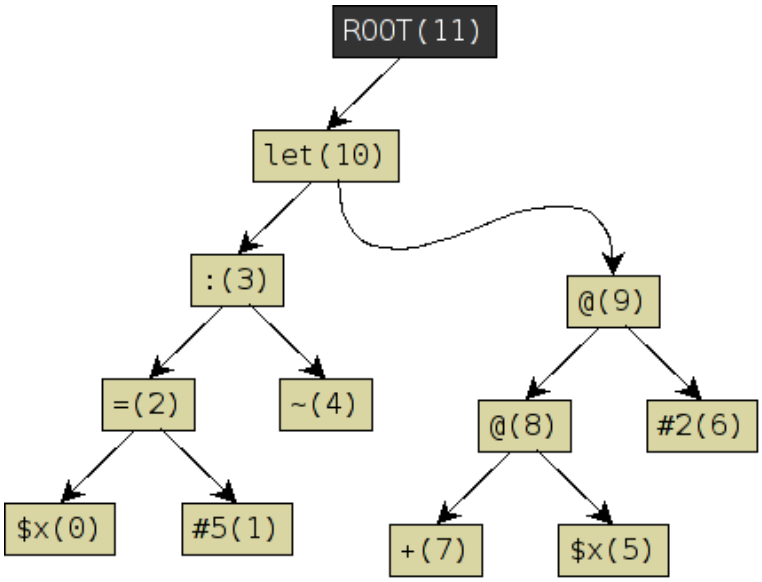

Figure 1: The graph after the parser stage. Notice that the let expression holds a list of assignments (in this case only one). The numbers in parentheses are the node numbers, which are like pointers to the node.

parser. Operator precedence follows rules much like those of other C-derived languages.

My language uses the `.` (dot) operator to represent function application. Most similar languages use whitespace to separate the arguments. By using a separate character, function application becomes an infix operation, which makes the syntax and parser much simpler. Almost all operations other than lambda definition, literal list specification, and control structures are represented as infix expressions. This is far more intuitive than the prefix notation of Lisp, and matches expressions in almost all imperative languages.

The language has support for the let construct, which allows local definitions, as well as variables in the imperative style. The language is lazy, which means functions are not evaluated until the results are needed. The current model only accounts for one level of "need", but future iterations will likely include support for "strong" and "weak" need. Built in operators and functions such as `+`, `->`, and `reduce` would 'strongly' need their arguments, while user defined functions would 'weakly' need their arguments. Closures would only evaluate themselves when they are strongly needed, but regular expressions would do so only on weak need. This would cause expressions such as

```
let
   two = {x,y | do y; x; () done}
in
   two.(print."x").(print."y")}
```

to print "xy" rather than "yx". This is what most imperative programmers would expect, as well as those used to Lisp-like semantics. If the function `two` had been called on two closures

instead, then the evaluation of the print statements would have been delayed until the code had entered the do construct, in which case the output would be "yx".

The do/done construct allows code to be executed in a definite order. This is useful for things such as print statements, and other functions with side effects. The presence of side effects in functions will be tracked, so that optimizations can be applied to functions which have no side effects.

Other similar languages focus on having a strong mathematical foundation. My language is not designed with mathematical elegance in mind, but rather with being a concrete language that is useful. Lisp uses the same representation to represent code and data. While this makes certain kinds of programming easier, it is much more difficult to optimize, because the original representation must be retained and the optimized version must be recreated when the code is modified.

4 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. It does this using a state machine. For each character, it determines whether it can be added to the existing token. If it can, then it is appended to the token string. Otherwise, the token is considered complete and it is added to the list of existing tokens. The lexical analyzer keeps track of line numbers and attaches them to tokens, which allows the parser to report errors accurately. It also parses the constants (such as numbers) to produce values, and un-escapes the strings to produce an in memory representation.

The parser is the next stage. The parser turns the linear token stream into the first revision of the graph that will eventually be executed as the program. The parser has many lines of code, but as it is a simple recursive descent parser, it is not terribly complex to understand. In the Python version, techniques were used to make the code smaller, in an attempt to make it more concise and readable.

The entry function for the recursive descent part of the parser is rootexpr(). This function first looks for a statement on the input token stream (such as let, if, or do). If one of these is found, then the code descends into the function for that construct. Such constructs always begin with a keyword, making them very easy to find. If a keyword is not found, then the expression must be the beginning of an infix expression. This requirement (keywords first) makes it easy to parse the language using the simple recursive descent parser. The infix expressions have a much more complex stack, because they must correctly handle operator precedence.

After the parser has constructed an in memory graph of the program, the optimizer is called to transform this graph. Despite it's name, the optimizer has a larger role than simply to make the code execute faster. 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` from their tree representation into the graph form. It also performs lambda lifting, to allow the code to be compiled into

a linear instruction stream. A compiler would produce its output at this stage, while an interpreter would continue on.

The executor is the final stage of program execution. The interpreter is responsible for walking the graph and performing the instructions found there. The interpreter uses what is known as normal order reduction to ensure that the program will complete if it will ever complete. This is used when there is a chain of application nodes in the graph. These nodes represent a curried function being applied to a series of arguments. The do/done construct needed its own special case code, to ensure it would execute in order. All of the mathematical operators are implemented in terms of internal Python operators.

The lambda-lifting operation allows this part of the system to run much faster. Without lambda lifting, any recursive function would have to duplicate a copy of itself for every invocation, which leads to exponential behavior. With the lifting process, the need to duplicate any function is eliminated. This leads to much faster performance. It can also open the door to more aggressive optimizations, and possibly machine code compilation.

The language requires the print statement to output any data. With the implementation of ordered evaluation, the language does not require monads or other obscure and complicated constructs to perform something as simple as streamed input and output.

5 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 user defined functions. Lists can be traversed using the `head` and `tail` functions. There is support for imperative programming using the do/done construct. Recursive functions are fully supported through the implementation of lambda lifting in the optimizer. 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. This allows the language to be used in the traditional file based way, as well as in an interactive manner.

6 Future Work

Future work on my language will include the further development of the core functions. The system still needs an implementation of weak and strong need, as well as a means on controlling this programmatically. The standard library needs to be developed, as well as bindings to common libraries. Example programs demonstrating how the language can be embedded in other languages would provide a convenient reference for other programmers. The optimizer can be improved to do things such as constant propagation, tail recursion optimization, and code compilation.