An Attempt to Write a Purely Functional Programming Language Optimized for a Concise Structure

Hunter Waldron

April 23, 2019

I attempted to write a new programming language to test several ideas that I have developed over years of programming and thinking about programming. I have not named my project, as not only am I incapable of imagining a suitable name, but I also believe that because there are a limited number of combinations of characters in the English language, there are a limited number of names, and it would be selfish to take one of them for a mere amusement such as this project. Therefore I will refer to my project as "this language" throughout this paper.

This paper discusses the motivation behind many decisions that were made when designing and implementing this language, and attempts to justify some of the less typical decisions. This is paper is not a manual, a catalog of the features this language supports, or a complete description of the implementation of this language.

The Implementation

I implemented this language in C because C has the perfect amount of abstraction for a project like this. I also wrote an interface for performing system calls directly so I did not have to rely on or deal with the poor and archaic design of the standard C library. I only supported modern Linux because that is the simplest and most powerful platform for software development I know of, and this is a research programming language that does not need to be distributed.

The programming language implementation is divided quite stringently into three layers, each of which is responsible for a particular stage of the program execution process. These layers communicate with each other in a single direction, as can be seen in figure 1.

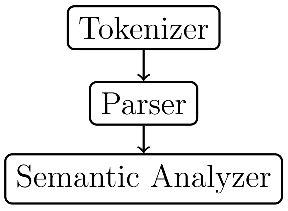

Figure 1: Interlayer communication

The tokenizer is the first layer, responsible for reading the source code directly. The tokenizer scans the source code for words, of which all relevant information is packaged into tokens. The second layer, the parser, receives these tokens and uses them to generate a syntax tree. Finally, the syntax tree is passed to the third layer, the evaluator. This layer executes the program by traversing the syntax tree and applying rules to every vertex. If no error occurs, and the language run time terminates, the program will have been executed.

The structure of a Program

Although there are many different ideas that together shaped the design of this language, the foremost design objective was to create a functional programming language in which programs can be written either as purely mathematical formulae, as a sequence of instructions, or any combination of the two. Furthermore, there was to be a simple but stringent means of encapsulating different components of a program so as to abstract their particular details from each other.

To create a language where these objectives can be properly implemented, we must first precisely define what a program is. A program in this language can be though of as a dependency graph where every edge has a one direction communication channel to the vertex it points to. In the simplest case this graph is a tree, where every vertex takes input from its parent vertex and depends on its child vertices. In a terminating program, every leaf vertex can be evaluated with only the input it is given, and so the entire program can be evaluated by passing information down the tree, then at every vertex, combining the calculations performed by child vertices. However, this is not every case. Where there is recursion there must be loops in the graph or even cycles where edges point back to higher vertices of the graph. Notice that, in this model of programming, there is no room for general IO. Input given to the program must be taken from the highest vertex. This keeps the language purely functional and easy to reason about at the cost of practicality. A small price to pay.

This thinking lead to the idea of what I call a “block expression”, which is a lexically scoped set of function definitions paired with a “return expression”. The block expression is evaluated by evaluating the return expression, which can be any expression containing references to the functions defined in the block expression, as well as at every lower level of the lexical scope. In this structure, each function definition can be considered to be an instruction, which can depend on the previous instruction, the next instruction, or even an instruction at a lower level of the lexical scope. The return expression tells the block expression which instructions to execute and how to combine their results to give back to the parent.

To distinguish different block expressions from each other, every block is required to have a homogeneous indentation level, and be at a higher indentation level than the parent block expression is. This sort of requirement may be a point of contention among many programmers, and makes this language context sensitive, but using indentation rather than, for example, opening and closing curly braces, keeps the screen clean of excess symbols.

In figure 2, an example of a simple function is given to demonstrate the expression block. In the figure, `f` is a function that is defined as an expression block containing two functions and a return expression. As can be seen in this example, the return expression refers to `a` and `b` and then combines their calculations.

```
f y :=
    a x := + x 1
    b x := * x 2
    return + (a y) (b y)
```

Figure 2: A simple code example

The same function `f` that was defined in figure 2 can be defined as a single mathematical formula, as is shown in figure 3.

The more observant reader will have noticed that the notation used to define `f` in both of these examples is Polish notation.

```
f y := + + y 1 * y 2
```

Figure 3: An equivalent definition of `f` from Figure 2

Notation

Solving the problem of how to allow the programmer to define their own operators was also considered and was to a lesser extent prioritized. When the programmer can define new operators and overload existing operators, a powerful but quite easy to understand and implement polymorphism is had, which is sometimes called ad hoc polymorphism by computer scientists. If used well, ad hoc polymorphism can allow much easier to understand programs. In the hands of a less competent programmer, however, ad hoc polymorphism can create obfuscation, the likes of which are not possible without operator overloading. But because only good programmers will ever use this language, that is no issue.

All of the following features were desired for user defined operators.

- The user can define any unary and binary operator for any types.
- The same operator can be used to define both a unary and a binary operator.
- The precedence of an operator with respect to every other operator must be unambiguously determinable.

Unfortunately, using conventional, infix notation, supporting all of these features simultaneously is a nightmare. Not only is there the problem of determining if an operator is unary or binary, there is also the problem of determining the precedence of an operator with respect to all others. And what if the programmer overloads an operator but wants the precedence of that operator to be different for each overloading? Either some compromise must be made, such as giving all user defined operators the same precedence, or a different approach than infix notation must be used.

Using Polish notation eliminates all of these problems. With Polish notation, there is no such thing as operator precedence, and the arity of an operator can be determined quite easily. Because there is no operator precedence, parenthesization is also unnecessary. And furthermore, Polish notation is easier than infix notation to implement in the parser.

The Parser

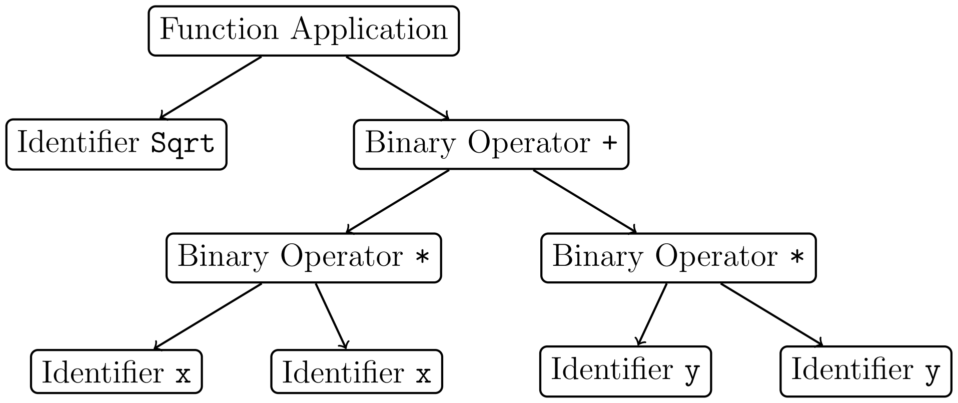

Figure 4: The syntax tree generated by the expression `sqrt + * x x * y y`

The parser was designed to be as simple as possible while still providing support for every language feature that was required. In some ways the parser can even be regarded as primitive. For example, there is no support for token lookahead, as token lookahead was never necessary. However, to properly handle the notation and the hierarchy that this language requires, expressions must be parsed at three different levels.

At the highest level, there are the block expressions, the purpose of which has already been explained. There are also full expressions and, at the lowest level, there are the simple expressions. Simple expressions are the smallest components of a computation. For example, function identifiers and literals are simple expressions. Full expressions consist of all simple expressions as well as function application.

Function application is parsed as a simple expression applied to another simple expression. This can be done repeatedly by placing several simple expressions next to eachother. For example, the full expression `f g h` is parsed as (`f` applied to `g`) applied to `h`. Function application can not be categorized as a simple expression, for if it were, then `f g h` would instead be parsed as `f` applied to (`g` applied to `h`).

Full expressions and simple expressions may be categorized differently, but they necessarily must be mutually recursive. For example, a parenthesized full expression must be categorized as a simple expression to allow an expression like `f (g h)` to be parsed as `f` applied to (`g` applied to `h`).

For an example of how a full expression is parsed, consider the expression that calculates the norm of a pair (x, y). In this language, assuming there is a function named `sqrt` which calculates square roots, this can be written as `sqrt + * x x * y y`. When parsed, this expression becomes the syntax tree shown in figure 4.

This example shows how Polish notation and function application can be implemented together rather nicely. No parentheses are needed for this example, but the expression is no less readable than the infix equivalent.

Formal Definitions

Full expressions are completely context free, and as so they are not terribly difficult to precisely define. In Backus-Naur form, full expressions can be defined as so:

```
<fexpr> ::= <sexpr> | <apply>
<apply> ::= <sexpr> <sexpr> | <apply> <sexpr>
<sexpr> ::= <fidnt> | <intgr> | <strng> | <unaop> |
            <binop> | <cntnl> | <tuple> | <ilist> |
            "(" <fexpr> ")"
<unaop> ::= <oidnt> <sexpr>
<binop> ::= <oidnt> <sexpr> <sexpr>
<tuple> ::= "(" <elements> ")"
<ilist> ::= "{" <elements> "}"
<elements> ::= <fexpr> | <fexpr> "," <elements>
<cndtl> ::= "when" <cndtns> <fexpr>
<cndtns> ::= <cndtn> | <cndtn> <cndtns>
<cndtn> ::= <fexpr> "then" <fexpr> "else"
<fidnt> ::= <fidntchar> | <fidntchar> <fidnt>
<oidnt> ::= <oidntchar> | <oidntchar> <oidnt>
<strng> ::= '"' <strngbody> '"'
<strngbody> ::= <strngchar> | <strngchar> <strng>
<intgr> ::= <intgrchar> | <intgrchar> <intgr>
<intgrchar> ::= "0" | "1" | "2" | "3" | "4" |
                "5" | "6" | "7" | "8" | "9"
```

This is, strictly speaking, an incomplete definition, as there is no expansion rule for `fidntchar`, `oidntchar`, or `strngchar`. There are two reasons these were omitted. First, the symbols that are used for each of these are arbitrary and irrelevant to understanding the full expression. Furthermore, the expansion rules that were actually used in the implementation allow a large number of symbols each, and thus will take a substantial amount of page space if they are written here.

Unfortunately, although this language is relatively simple, the entirety of it cannot be defined using Backus-Naur form, as block expressions are sensitive to the indentation level. If that weren't so, and instead, for example, opening and closing curly braces were used to start and terminate blocks, then the entire language will be context free and thus definable using Backus-Naur form.

From these definitions, the structure of a syntax tree representation of a full expression can be seen. When the parser runs, many instances of this structure will be placed in memory, along with the similar structure that is used for block expressions.

Memory Management

There are two problems that become apparent when contemplating the memory requirements of the parser, both of which may be solved with using a single solution.

The parser must make a substantial number of small memory allocations in many different places across every parser subroutine. In each of these subroutines, there are many possible errors that can occur, none of which may be ignored. When an error occurs, the parser can only relinquish the work that it was assigned and, before notifying the caller of its failure, it must deallocate every resource that it ever allocated. Requiring that, if an error occurs, each of the parser subroutines deallocate all resources that they have allocated will certainly ensure that this happens. However, such a strategy requires that no small amount of additional complexity be added to each parser subroutine, for each parser subroutine may be in any of a large number of different states with regards to memory allocation when an error occurs, and each of these possible states must be considered. The likelihood that one of them is either ignored or mishandled may be high.

The amount of memory being allocated must also be considered. If memory is allocated to the parser in such a way that every individual memory allocation may be individually deallocated and reused elsewhere, then every memory allocation must have associated with it the amount of memory that

that particular allocation occupies. Although this functionality is not difficult to add to a memory allocator, and for each individual memory allocation the added overhead will be small, because the parser will allocate many small portions of memory, the added overhead will substantially increase memory usage overall.

Both of these problems are solved by using a different memory allocator for every instance of the parser. Each memory allocator has its own memory from which small portions are allocated to its parser. If the parser encounters an error, instead of deallocating each individual allocation, the entirety of the memory can be deallocated at once. Otherwise, if the parser succeeds, what memory that was not used may be given to another memory allocator, and none of the memory that was allocated need ever be deallocated. This also eliminates the need to track the size of each memory allocation, and thus solves both problems.

The Result

For the most part, I consider my project a success, as not only did I succeed in creating a new programming language, but several of the ideas I incorporated into the language show the same promise that I had hoped they would. For example, I believe that Polish notation has merit and deserves to be considered for serious, industry programming languages. I do not consider my work to be a complete success, however, as not only did I fail to include every feature into the programming language that I wanted, but I also found that some of my ideas were, to an extent, ill conceived. Moreover, this language lacks a proper type system, and fails to detect many detectable errors that can occur before runtime, although that is only due to the limited amount of time I had. Be that as it may, I gained much insight into programming language development and design through working on this project that I otherwise wouldn't have.