Technical Report

UCAM-CL-TR-89
ISSN 1476-2986

Number 89

Making form follow function

An exercise in functional programming style

Jon Fairbairn

June 1986

ISSN 1476-2986

Making Form Follow Function

An Exercise in Functional Programming Style

Jon Fairbairn

University of Cambridge Computer Laboratory

Abstract

The combined use of user-defined infix operators and higher order functions allows the programmer to invent new control structures tailored to a particular problem area.

This paper is to suggest that such a combination has beneficial effects on the ease of both writing and reading programmes, and hence can increase programmer productivity. As an example, a parser for a simple language is presented in this style.

It is hoped that the presentation will be palatable to people unfamiliar with the concepts of functional programming.

1. Introduction

It is widely accepted that the most important aspects of the task of programming are the design of data structures and of algorithms. It is the intention of this paper to suggest that the design of notation can be a part of programme design, and an important part at that. A secondary objective of this paper is to present some functional programming techniques in a manner acceptable to a wider audience.

I hope to demonstrate that these techniques improve the readability and writability of programmes. In this style of programme development the programmer is at liberty to define what amount to new control structures, tailored to the problem at hand. The net effect can be to make the programme very close in appearance to its specification, or, to make the form of the programme follow its function.

The central notions are higher order functions, user-defined operators and the use of (lazy) lists to represent input streams. Although few conventional languages permit all of these, the style of programming may still be useful to the imperative programmer. Unfortunately most conventional languages would not permit the neat structure that is used here.

The example I have chosen for this paper is the writing of parsers, a task for which this style of programming is particularly suitable, but a similar method has been applied with some success to programmes dealing with graphical structures [Henderson 1982] and to a small simulator for logic gates. The parser will be for a simple language with the following grammar:

```
<programme> ::= <declaration> ; <programme>
              | <assignment> ; <programme>
              | <command>
<command> ::= <assignment> | <expression>
<declaration> ::= Declare <name>
<assignment> ::= <name> ← <expression>
<expression> ::= <term> + <expression>
               | <term> − <expression>
               | <term>
<term> ::= <factor> × <term>
         | <factor> ÷ <term>
         | <factor>
<factor> ::= <name> | <integer>
           | (<expression>)
```

Fortunately this gives us a clear, regular description of the problem. The exercise was to convert this description into a programme. With the suitable definition of operators for succession (▷), alternation (|) and an operator as to introduce semantic interpretation functions, a parser for this grammar can be written in a functional language like this:

```
Letrec programme ≜ (declaration ▷ literal ";" ▷ programme   as build-sequence)
                 | (assignment ▷ literal ";" ▷ programme    as build-sequence)
                 | command,

       command ≜ assignment | expression,

       declaration ≜ literal "Declare" ▷ name   as build-declaration,

       assignment ≜ name ▷ literal "←" ▷ expression   as build-assignment,

       expression ≜ (term ▷ literal "+" ▷ expression   as build-infix)
                  | (term ▷ literal "−" ▷ expression   as build-infix)
                  | term,

       term ≜ (factor ▷ literal "×" ▷ term   as build-infix)
            | (factor ▷ literal "÷" ▷ term   as build-infix)
            | factor,

       factor ≜ ((name | integer)   as build-factor)
              | (literal "(" ▷ expression ▷ literal ")"   as parenthesised-expression)
```

2. Explanation

This is a representation of a recursive descent parser. The notation used is essentially the applicative language Ponder [Fairbairn 1983] [Tillotson 1985], but the example would be much the same in any other functional language that permits the definition of infix operators. Letrec begins a series of mutually recursive definitions, each of the form <name> $\triangleq$ <value> (the symbol $\triangleq$ means "equals by definition"). So a name is defined in the programme for each of the symbols of the grammar. It is assumed that functions that recognise terminal symbols such as *name* and *integer* have already been defined.

Although the programme closely resembles the grammar, there are differences. The most obvious difference is that succession is represented by juxtaposition in the grammar and by $\triangleright$ in the parser. The next difference is that the programme contains more information than the grammar — a programme that simply responded "That matches the grammar" or "that doesn't match the grammar" when presented with some sample input would be of limited use. As well as how to parse, the programme also tells what to do with the result: functions such as *build-sequence* are applied (by the operator as) to the results of parsing the sub-expressions, in order to produce the result of parsing the combined expression.

The idea is simply that the operators $\triangleright$ and | will take parsers as arguments and produce parsers that accept the strings belonging to their succession and alternation respectively. The basic idea is similar to that of Burge [Burge 1975], but what makes the difference is the effect that infix operators have on the appearance of the programme: $a \mid b \mid c \mid d$ is much easier to read than $or\ a\ (or\ b\ (or\ c\ d))$.

In this implementation a parser is represented as a function that takes a list of tokens and produces either an indication of failure or a parse tree and a list of the tokens remaining on the input. So if *parser* accepts <tok$_1$> <tok$_2$>, then

$$parser\ (tok_1{::}tok_2{::}tok_3{::}\ldots) \Rightarrow ((tok_1, tok_2), (tok_3{::}\ldots))$$

(:: is the list construction operator), otherwise

$$parser\ (tok{::}\ldots) \Rightarrow nil$$

(*nil* is the result of parsing something that doesn't match).

Now we can see how to write the function that parses alternatives: $parser_1 \mid parser_2$ will produce a new parser that tries $parser_1$ on the input and if that fails, tries $parser_2$. So | is a function that takes two parsers and produces a third. One could imagine writing such a function in a conventional language:

```
Proc or (parser_1, parser_2) =
      Begin Proc parser_3 (input) =
                  If parser_1 (input) succeeds
                  Then parser_1 (input)
                  Else parser_2 (input)
            Return parser_3
      End;
```

Of course, this would not work in most languages, because of restrictions on returning procedures from procedures, but I think it illustrates the general idea. Notice that the input on which a parser operates is passed as a parameter. The absence of side effects in functional languages ensures that when $parser_3$ finds that $parser_1$ fails, the *input* that is passed to $parser_2$ is precisely as it was before $parser_1$ was called.

When a parser succeeds the result returned is the parse tree together with the *remaining* input. So $parser_1 \triangleright parser_2$ just applies $parser_1$ to the input and if it succeeds applies $parser_2$ to the remainder of the input that is returned as part of the result of $parser_1$.

3. Advantages

The programme given above is just a recursive descent parser written in a functional language. It would have been possible to write a programme that performed the same function in a more conventional manner, defining procedures to parse each of the grammar items in terms of explicit tests on the input. The method used here has the immediate advantage that the correspondence between the programme and the grammar that it parses is obvious. This means that the effort needed to explain the programme (to someone familiar with functional programming, at least) is much less, so fewer comments are needed in the code. This in turn means that the modification (and keeping comments in step with modification!) is much easier.

The example is perhaps less than perfect in that parser generators already confer most of the suggested advantages, but it must be remembered that I only chose the example to be a problem familiar to most programmers. Notwithstanding that, I have found this method of writing parsers to be an improvement over the use of parser generators in a number of ways. The first is that it is more direct — the interface with the rest of the compiler is under the control of the programmer. The more important advantage is that the parser can be extended beyond the power of most parser generators, to parse languages that permit syntactic extensions, for example.

4. Drawbacks

The picture painted so far is rather rosier than reality. Although the correspondence between the grammar and the programme for this example is almost exact (I have omitted the data-type information — not all functional languages require this anyway), some grammars are not so easy to translate as one might hope.

The most serious problem is that the programmed | operator does not behave in precisely the same way as the operator in grammars. A programme containing $parser_1$ | $parser_2$ will (as described) try $parser_1$ on the input and then try $parser_2$ if $parser_1$ fails. The snag is that if $parser_1$ succeeds on an initial substring of a larger expression accepted by $parser_2$, only $parser_1$ will be tried†. This means that rules such as <expression> must be coded with the longest rules first. A better alternative is to 'factor out' the offending substring (see below).

The other problem arises when the grammar contains left-recursive rules. The naïve coding of such a rule as

```
<l> ::= <l> <r> | <s>
```

† It is possible to define | so that it tries both, and returns a list of successful parses, as in [Wadler 1985], but this method can result in parsers that are very inefficient because they parse initial strings several times. Once these inefficiencies have been optimised out, the parsers look almost identical to the ones suggested here.

will begin with a recursive call to the parser for <l>, which in turn begins with a recursive call to the parser for <l>, and so on *ad infinitum*. Careful analysis of the rule will show that an <l> is an <s> followed by an arbitrarily long sequence of <r>s. What is needed to solve the problem of infinite recursion is to re-code the production for <l> so that it says just that. We can then write an operator that parses such sequences, so that the parser looks like

$$l \triangleq s \ \textbf{then-an-arbitrary-sequence-of} \ r.$$

I think that this is clearer anyway (although a shorter name for the operator might be preferred).

An alternative coding of this would be to introduce a new grammar operator *, as is often done, and write <l> as <s> <r>*, which produces the same set of expressions as before, but with different parse trees. It is not difficult to write a function for *.

Similarly a notation for optional grammar items can be introduced, allowing *expression* to be coded as

$$expression \triangleq term \rhd [(\textbf{literal} \text{ "+"} \mid \textbf{literal} \text{ "−"}) \rhd expression\ ] \quad \textbf{as} \ build\text{-}infix$$

mimicking the extended BNF notation <term> [(+|−) <expression>]. This overcomes the problem of parsing the initial <term> twice.

5. Extensions

As well as the option and arbitrary repetition operators mentioned above, other extensions to this method prove useful when dealing with more complicated grammars.

If *attributes* of the parsed sub-expressions are included in the data upon which the parser works, parsers for attribute grammars may easily be written. A simple example of this would be to code the binding powers of the $+$, $-$, $\times$ and $\div$ as integer priorities. The parser could then parse an expression of priority n, being either an expression of priority $n-1$ or two such expressions separated by an operator of priority n. This would not make the present example any shorter, but would have obvious benefits if the grammar included relational operators and so on, each at a different level of priority.

Similarly higher level grammars may be simulated by writing new operators for list-of-thing and such like to correspond to the meta-rules.

6. Lexical Analysis

Lexical analysers can also be written in this style, by giving a definition of each lexeme in terms of its characters, and then defining a lexeme to be the alternation (using |) of all the distinct lexemes, and then using * to make them into a list.

For instance we might code a parser for a <name> as:

$$name \triangleq letter \rhd (letter \mid digit)^* \quad \textbf{as}\ build\text{-}name$$

7. Comments

- The only real reliance on lazy evaluation and the freedom from side effects is in the use of the lazy list of tokens for the input. This kind of lazy list can be modelled fairly easily in a decent imperative language, by the use of explicit suspension functions.
- The operators $\rhd$ and | are useful for things other than parsing — any task that can be solved by successive applications of methods, or by one of a number of methods, can be attacked with these operators. Compare the *tacticals* used in LCF [Paulson 1985].
- Although this technique relies heavily on higher order functions, the run-time penalty is minimal because applications of operators such as $\rhd$ can be expanded out at compile time. This, of course, suggests that the operators might be written using macros in a conventional language.

Bibliography

[Burge 1975]:

W.H. Burge

Recursive Programming Techniques

Addison Wesley 1975

[Fairbairn 1983]:

Jon Fairbairn

Ponder and its Type-system

Technical Report 31, University of Cambridge Computer Laboratory, 1983

[Henderson 1982]:

P. Henderson

Functional Geometry

in 1982 ACM Symposium on Lisp and Functional Programming, August 1982

[Paulson 1985]:

Lawrence C. Paulson

Interactive Theorem Proving with Cambridge LCF — A Users Manual

Technical Report 80, Cambridge University Computer Laboratory, November 1985

[Tillotson 1985]:

Mark Tillotson

Introduction to the Functional Programming Language Ponder

Technical Report 65, Cambridge University Computer Laboratory 1985

[Wadler 1985]:

Philip Wadler

How to Replace Failure by a List of Successes

In Lecture Notes in Computer Science 201, Springer Verlag 1985