Using Functional Parsing to Achieve Quality in Software Maintenance

(IFI TR 93.36)

Peter Baumann Jürg Fässler Markus Kiser Zafer Öztürk *

Institut für Informatik der Universität Zürich,
Winterthurerstr. 190, 8057 Zürich, Schweiz

Email: {baumann,faessler,kiser,oeztuerk}@ifi.unizh.ch
Telephone: +41-1-257 4307. Fax: +41-1-363 0035.

Keywords: software maintenance, software quality, functional programming, functional parsing, incremental parsing, Standard ML.

Abstract: Many software maintenance tools rely on a parser. When integrating several maintenance tools into a software maintenance environment, it is useful that all these tools are based on the same flexible parser. In this paper, we present a method of implementing such a parser in a functional way.

A functional parser can be built using basic parsers for the terminal symbols, and non-basic parser for the non-terminal symbols of a grammar. To create the latter, so-called parser combinators – implemented as higher-order functions – are used. It turns out that by using the functional parsing approach, large parts of a parser can in fact be generated automatically.

The resulting functional parser has some very useful features which make it ideal for use in a software maintenance environment. The most important one is that the parser is incremental in its nature, which means that we automatically obtain a parser for every single production of the underlying grammar in EBNF. Moreover, due to this one-to-one correspondence between the parser and the EBNF productions, there is a very clean and consistent way to use the parser as a front-end for tools to be developed on top of it.

We have implemented a functional parser for COBOL 74 in only a few weeks. We will use this parser for all the tools in the software maintenance environment *AEMES*[1] which is currently being developed at the Institut für Informatik of the University of Zurich.

**Jürg Fässler* is supported by KWF (Schweizerische Komission zur Förderung der wissenschaftlichen Forschung), grant nr. 2326.1 *Zafer Öztürk* is supported by Swiss Life Insurance and Pension Company.

[1] *AEMES* stands for "**A***n* **E***xtensible* **M***aintenance* **E***ngineering* **S***ystem*", a software maintenance project whose aim is to integrate a collection of maintenance tools into a consistent environment. This project is supported by *KWF*, *Swiss Life Insurance and Pension Company* and *Bull*.

1 Introduction

It is well known that software maintenance is an important part of the software life-cycle. Software maintenance is the key issue in order to preserve a software's operability, to adjust a software product to new requirements, and to improve its quality.

But software maintenance is difficult and time-consuming and is thus a costly issue. Some authors claim a number as high as 80% of the overall cost for the development of software products to go into the maintenance [Hal92, pp.218ff]. The time spent for software maintenance is usually said to be between 50% and 75% of the overall time of the life-cycle of a software system [Som92, pp.534].

These numbers clearly show that it is very important to use advanced technologies for creating powerful software maintenance tools. After all, the software maintenance tools should be easily maintainable themselves and not create another maintenance problem.

Maintenance tools normally rely on a parser for the language of the programs to be maintained. The parser is therefore one of the most crucial components of maintenance tools or of maintenance environments. The more flexible the parser is, the easier it is to create a rich set of maintenance tools on top of it.

In this paper we propose an approach for creating such a parser for software maintenance tools in a functional way. To the best of our knowledge, no similar approach has ever been used to develop a software maintenance environment. This is even more surprising since our approach has some striking advantages over a more conventional approach.

With the help of this technique, we were able to derive a grammar in EBNF for the COBOL 74 standard [ANSI74] and to develop a parser for this grammar. This was achieved within a few weeks. We expect that our parser is powerful and flexible enough to be used as a generic front-end for creating software maintenance tools. They will later be integrated into our software maintenance environment *AEMES* [FKO92].

Our approach of implementing a parser has several advantages over more conventional approaches using e.g. *lex* and *yacc*. Large parts of our parser can be generated automatically from a grammar given in EBNF. Therefore, it becomes very simple to create new, modified, or improved versions of a parser. This flexibility proved to be particularly useful for the development of a parser for COBOL. A grammar for COBOL is large, hard to derive from the ANSI standard and there exist several vendor-dependent dialects. But we would like to stress that the functional approach we present here is in no way restricted to a particular language. A functional parser can be developed for any language for which there exists a grammar in EBNF.

In the following section we will give a brief introduction to the principles of functional parsing. It is followed by a section describing the use of functional parsing in software maintenance. This section also encloses some detailed examples. We used Standard ML as the implementation language, which is one of the most advanced functional programming languages of today. The fourth section contains some remarks regarding practical aspects of functional parsing. We are closing with conclusions and an outlook on our future work.

2 Functional Parsing

This section gives a brief introduction to the principles of functional parsing. More detailed descriptions of functional parsing can be found in [Rea89, Pau91, Hut92].

The key idea behind functional parsing is that parsers are *functions*. They take as input lists of tokens and produce as output some kind of internal representation of the parsed tokens. Each production of a grammar in EBNF is represented by such a parsing function. A functional parser for a language is then a combination of the parsers of its productions. The parser is built using two different classes of parsing functions:

- *basic parsers* for each terminal symbol (token) of the grammar, and
- *non-basic parsers* for non-terminal symbols which are formed from other parsers by so-called parser combinators.

The input for a parser is produced by a lexical analyzer, which is itself a function. This function goes from the source codes of programs to lists of tokens.

2.1 The lexical analyzer

The function `lexer` representing the lexical analyzer takes as input a character stream that is simply the source code of a program. The output of the `lexer` is a token list representation of the program. Thus, `lexer` has the following signature:

`lexer:` `programs` $\rightarrow$ `lists of tokens`

The function `lexer` must be able to recognize all terminal symbols (tokens) in the program sources. It can normally be written in a straight-forward way for a given grammar.

2.2 Functional parsers in general

As already mentioned, a parser takes a list of tokens as input. It consumes as much as possible from the list of tokens and returns an internal representation of the part it was able to recognize together with the rest of the list that has not been parsed.

Because parsers are functions, they can easily be implemented in a functional programming language like Standard ML. The signature of a parser looks like:

`parser:` `lists of tokens` $\rightarrow$ `internal reps` $\times$ `lists of tokens`,

where `internal reps` stands for the set of internal representations, e.g. parse-trees.

When combining the two functions `lexer` and `parser` in a functional way, the resulting composed function gets the following signature:

`parser` $\circ$ `lexer:` `programs` $\rightarrow$ `internal reps` $\times$ `lists of tokens`

2.3 Basic parsers

For each terminal symbol in a grammar, the functional parsing approach requires a separate parser. We call these parsers *basic parsers*. Usually, such basic parsers encompass parsers for keywords, identifiers, numbers and so on. They all simply check if the head of the list of tokens equals the expected token.

As a small example, let's assume we have an assignment statement of the following form:

```
answer := 42
```

For this statement, `lexer` returns the following list of tokens:

```
[ident "answer", keyword ":=", number "42"]
```

Feeding this list of tokens to the identifier parser `parse_ident`

```
parse_ident ([ident "answer", keyword ":=", number "42"])
```

leads to the following result

(*internal rep of* `ident "answer", [keyword ":=", number "42"])`

The parser simply recognizes the identifier `"answer"` and returns an internal representation for the identifier `"answer"` as well as the unparsed rest of the list of tokens.

Basic parsers are easy to implement in the functional approach and the program code for all of them looks very similar. Furthermore, the number of basic parsers is usually not very high, e.g. for our COBOL parser, we had to write only 9 basic parsers.

2.4 Non-basic parsers and parser combinators

Non-basic parsers are used for parsing non-terminal symbols of the grammar. Non-terminal symbols are defined using grammar productions. In such productions there exists a set of possibilities to combine terminal and non-terminal symbols to obtain new non-terminal symbols. These possibilities are:

1. $a = b\ c$.
   This production defines a in terms of b followed by c. To make this combination more explicit, we write $a = b\ \&\ c$.

2. $a = b \mid c$.
   This production defines a as being the alternative of either b or c.

3. $a = [\ b\ ]$.
   This production defines b as optional in the production for a.

4. $a = \{\ b\ \}$.
   This production defines a as the repetition of 0 or any finite number of b's.

5. $a = (b \mid c) \ \& \ d$.
   Parenthesis are used to express precedence.

If we would like to keep our functional parser representation as close as possible to the EBNF notation, we have to find a way to express these five combining possibilities in terms of functions. The idea is that these combinator functions will take parsers as their input and return a new (combined) parser as their output.

As we regard parsers as functions, these combinator functions must then be so-called higher-order functions. Higher-order functions can have functions as their input and/or produce functions as their output. Fortunately, such functions can be expressed very easily in a functional programming language such as Standard ML.

We implemented such higher-order functions, which are traditionally called *parser combinators*. As an example, let's have a look at the parser combinator &, which we write as `&&` in Standard ML:

```
infixr 4 &&;
fun (parser1 && parser2) token_list =
   let  val (x, token_list2) = parser1 token_list
        val (y, token_list3) = parser2 token_list2
   in
        ((x,y), token_list3)
   end;
```

Let's see how this works: The function `&&` is defined as an infix operator. It takes as input two parsers (`parser1` and `parser2`). `&&` first applies `parser1` to the input argument `token_list`. Then `parser2` is applied to `token_list2`, which is part of the output of `parser1`. At the end the results are combined and the remaining list of tokens `token_list3` is also returned. The new parser `(parser1 && parser2)` therefore takes as input `token_list` and produces as output the pair `((x,y), token_list3)`.

We can also give the signature of the function `&&`:

`&&:  parser_type` $\times$ `parser_type` $\rightarrow$ `parser_type`

where `parser_type` is an abbreviation for

`lists of tokens` $\rightarrow$ `internal reps` $\times$ `lists of tokens`

The other parser combinators can be defined in a similar way. All the details can be found in [Rea89, Pau91, Hut92].

With the parser combinators just described, we are now in a position to compose new parsers based on existing ones according to the rules inherent to the EBNF notation. In this way, we can obtain a parser function for each non-terminal symbol of a grammar. The parsing function for the starting symbol of a grammar is at the same time the parsing function for the whole language defined by a grammar.

It is important to note that the parser combinators are generic in the sense that they have to be written only once and can be used for building parsers for every language with an EBNF representation of the grammar.

One of the most appealing properties of the functional parsing approach is that in building a parser for the whole language, one gets fully functional parsers for each grammar symbol of the language for free. This allows great flexibility in parsing the source code.

3 Using Functional Parsing in Software Maintenance

In this section we point out the reasons why we think that functional parsing is of great help in software maintenance.

3.1 Creating Functional Parsers

One of the main properties of functional parsing is the one-to-one correspondence between the EBNF notation of a grammar and the implementation of the corresponding functional parser. For example, let us consider an EBNF production for the if-statement of a sample programming language which we define as follows:

```
if_statement = "IF" expression
               "THEN" statement_sequence
               { "ELSIF" expression
                 "THEN" statement_sequence
               }
               [ "ELSE" statement_sequence ]
               "END" .
```

The EBNF production above can be easily translated into a Standard ML function. We simply apply the basic parser `parse_keyword` and the parser combinators `&&`, `repetition`, and `option`. Additionally we use the parser functions for other non-terminals of the grammar which, in our example, are `parse_expression` and `parse_statement_sequence`. Note, that these two functions are implemented according to the same principles.

As the result we get:

```
fun parse_if_statement token_list =
    ( parse_keyword "IF" && parse_expression &&
         parse_keyword "THEN" && parse_statement_sequence &&
         repetition (
              parse_keyword "ELSIF" && parse_expression &&
              parse_keyword "THEN" && parse_statement_sequence
              ) &&
         option (
              parse_keyword "ELSE" && parse_statement_sequence
              ) &&
         parse_keyword "END"
         semantic_hook
    ) token_list ;
```

The function *semantic_hook* is the interface to the back-end of the parser. It determines the output of the parser which is usually some kind of internal representation (e.g. parse tree) of the parsed program. If the *semantic_hook* function is omitted, the parser will simply act as a recognizer for the language of the underlying grammar.

As one can see, transformation rules between EBNF notation and Standard ML notation are very simple and straightforward. Therefore, it is very easy to translate an EBNF grammar into a functional parser implemented in Standard ML. Because the transformation always follows similar patterns, transformation can be done automatically.

These properties are of great importance to build a flexible parser which can be easily adapted to different dialects of a programming language or even to different programming languages. Especially for languages like COBOL, this simplicity and flexibility when creating parsers is very valuable. Firstly, the grammar of COBOL is fairly large and can hardly be derived correctly from the standard at the first strike and therefore must be revised and refined several times. Secondly, even though COBOL 74 is standardized, there exist many vendor-specific dialects. With our approach, we are able to adapt our functional parser to different dialects simply by modifying the corresponding grammar in EBNF. And thirdly, our approach of building a parser easily allows to build a parser for any language whose grammar is available in EBNF. In fact, in our project so far the most difficult and time-consuming task was not to build the COBOL parser but to develop an EBNF for COBOL 74.

Grammars which can be expressed in EBNF notation belong to the class of context free grammars. In principle, a parser which is built according to the functional approach presented here is able to parse any formal language in the class of context free grammars. Therefore, we are not restricted to special subclasses of context free grammars like LL(k), LR(k), LALR, etc, which makes it easier to develop grammars and parsers for languages.

3.2 Incremental Parsing

Let us test the `parse_if_statement` function just defined. We can do so by writing in Standard ML:

```
(parse_if_statement ∘ lexer) ("IF a > 0 THEN a ELSE −a END")
```

Here, the output of `lexer` is used as the input of the function `parse_if_statement`. In fact the two functions are combined using the ∘ operator of Standard ML to build a new function. The signature of ∘ is:

```
∘:  (a → b) × (c → a) → (c → b)
```

where `a`, `b`, and `c` can be of any type. Note, that ∘ is also a higher-order function.

The result of applying the combined function to the input string thus is:

(*internal rep. of the if-statement*, `[]`)

The empty list `[]` of the result indicates that the whole input string could be successfully parsed.

As another example consider:

```
(parse_expression ∘ lexer) ("a > 0 THEN ...  ")
```

which returns:

(*internal rep. of the expression*, `[ keyword "THEN", ...  ]`)

Here the parsing was not exhaustive, since the part following `"a > 0"` is not part of an expression and could therefore not be recognized.

An example for an unsuccessful parse is:

```
(parse_if_statement ∘ lexer) ("IF a > THEN a ELSE −a END")
```

returning:

(*empty*, `[ keyword "IF", ident "a", ...  ]`)

by which the parser indicates failure. The reason is that between `IF` and `THEN` no valid expression could be detected. These few examples already make it clear how single parts of the parser can be tested separately. This is very handy for developing and testing a parser.

Another advantage of the incremental property of a functional parser is the ability to parse only parts of huge programs. In software maintenance, very often only small pieces of a program are affected by a change. Therefore, it is not necessary to parse the whole program but only the changed sections. This is achieved by trying different non-terminal parser to the changed code until a successful parse has been found. In practice, if the number of non-terminal parsers is very large some heuristics has to be found to determine which parsers to apply and in what order.

3.3 Extensibility of Functional Parsers

We have built the full parser by simply combining the function `lexer` with the parser function for the start symbol of the EBNF grammar:

`(parser ∘ lexer)`

It is natural to use the composition operator $\circ$ to add additional functionality to the parser. For example, if we extend a functional parser to a real compiler, we have to implement a code generator function `codegen` that takes as input the internal representation generated by the parser function and produces as output the appropriate object code. Once we have written the function `codegen` we can combine it like:

`object_code = (codegen ∘ parser ∘ lexer)` *source_code*

to obtain a fully operational compiler.

If we want to have a pretty printer or a cross referencer, we just have to write the corresponding functions and can create the pretty printer and the cross referencer in the same way as the code generator. Thus

`pp_source_code = (pprint ∘ parser ∘ lexer)` *source_code*

and

`xref_list = (xref ∘ parser ∘ lexer)` *source_code*

become fully functional programs. Many more examples can be found here. However, we are not restricted to always use `lexer` and `parser`. We can, for example, also write:

`simplify = (simplify_exp ∘ parse_expression ∘ lexer)` *source_code*

and

`datadep = (find_dep ∘ parse_expression ∘ lexer)` *source_code*

where we define two tools that solely rely on parts of the grammar and only parse specific expressions. In the example, `simplify_exp` is a functions that rewrites expressions in a simpler way and `find_dep` is a function that finds data dependencies in expressions.

Using the $\circ$ operator when creating tools, each tool consists of a sequence of processing steps. These processing steps have no side effects and are separated by well defined interfaces, namely the output type of one step must match the input type of the following step. Each one of the phases can be considered as a black box, which makes the implementation of tools very neat.

In summary, functional parsing lays the foundation to build a very powerful maintenance environment.

4 Some Practical Aspects

This section contains some remarks about the practical implementation of a functional parser.

A parser that has been built using the principles just described is not able to deal with *left-recursion* in a grammar. The parser would loop infinitely by repeatedly calling itself, without consuming tokens from the list of tokens. Fortunately, any left recursive production of an EBNF grammar can easily be transformed into a set of non left-recursive productions (see [ASU86, pp.176f] for more details). Thus, one can write a tool that automatically performs this transformation on the grammar before generating the Standard ML code for the parser.

Another aspect that needs some special attention is *cyclic definitions* in the grammar. Clearly, cyclic definitions lead to mutually dependent function definitions in the code of the parser. Depending on the language used for writing the parser, some form of forward declaration has to be introduced into the code for the parser. A particularly elegant way to deal with cyclic definitions is offered by the language Standard ML. There, recursive dependent functions can be defined in a natural way.

There is also a very nice possibility of *bootstrapping* the process of creating a functional parser. One does just have to write the EBNF grammar of EBNF itself. Taking the *Meta-EBNF*, the very same principles as shown in this paper can be used to create a parser for EBNF. A code generator can then be added to this parser, which constructs the Standard ML translation of a given grammar in EBNF for the considered language.

We would like to stress once again that our approach of building the parser has been proved to be particularly useful for debugging the grammar.

- Firstly, due to the incremental properties of the resulting parser, each production of the grammar can be tested separately.

- Secondly, it is easily possible to extend the capabilities of the parsers to find all possible parses instead of only one. Such an extended parser is very helpful to detect ambiguities in a grammar.

5 Conclusions and Future Work

Using the functional approach, we have been able to build a parser for COBOL 74 in very little time. Since large parts of our parser can be built automatically, relatively little time was spent on explicitly writing code for the parser. Most of the time was spent on writing and improving the EBNF-grammar for COBOL 74. Since every production of the grammar becomes a separate parser in our approach, testing of the grammar is greatly simplified.

We found that our approach is feasible enough to cope with a language like COBOL 74, whose grammar consists of more than 150 productions in EBNF notation. So we consider ourselves as being far from just having created a toy parser or an academic example.

We plan to incorporate our parser into the software maintenance environment *AEMES* we are currently developing at the Institut für Informatik of the University of Zurich. The aim of *AEMES* is to provide a consistent environment for several software maintenance tools which closely work together and whose data is put into a central store. We will use our parser as the front-end tool for each of the maintenance tools that become a part of *AEMES*. Thus, our next step will to create different back-ends for our parser. We will start by implementing the semantic hooks (see section 3.1).

Last but not least we would like to note that our experience with Standard ML has been very positive. Standard ML is, although a functional language and thus memory intensive, still reasonably fast. The pattern matching capabilities and the module system of Standard ML allowed us to write very readable and clean code. Furthermore, the polymorphic type system and the static type type checking capability of Standard ML have proven to be of great help for obtaining error-free program code.

We are convinced that our approach to developing a software maintenance environment is a concrete contribution to technology transfer and to the problem of making software maintenance more maintainable.

We would like to thank the Schweizerische Kommission zur Förderung der Wissenschaftlichen Forschung KWF in Berne, Bull France, Paris and Swiss Life Insurance and Pension Company, Zurich for supporting this project. We would also like to thank Prof. L. Richter for his encouragement and constructive criticism on this work.

References

[ANSI74] American National Standard Institute. *Programming Language COBOL*, X3.23-1974, ISO 1989-1978. American National Standard Institute, 1430 Broadway, New York, New York 10018, May 1974.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[FKO92] J. Fässler, M. Kiser, and Z. Öztürk. *AEMES* – first specification. Internal Report KWF2326.1(AEMES)-D01-SPEC V1.0 (12.92), Institut für Informatik, University of Zurich, Winterthurerstrasse 190, 8021 Zurich, December 1992.

[Hal92] P.A.V. Hall, editor. *Software Reuse and Reverse Engineering in Practice.* Chapman and Hall, London, 1992.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[Pau91] Laurence C. Paulson. *ML for the working programmer.* Cambridge University Press, Cambridge, 1991.

[Rea89] Chris Reade. *Elements of functional programming.* Addison-Wesley, Wokingham, 1989.

[Som92] Ian Sommerville. *Software engineering.* Addison-Wesley, Wokingham, England, 4th edition, 1992.