Using Standard ML in Software Maintenance: A Promising Approach

(IFI TR 93.37)

Peter Baumann, Jürg Fässler, Markus Kiser, Zafer Öztürk, Lutz Richter [1]

Institut für Informatik der Universität Zürich,
Winterthurerstr. 190, 8057 Zürich, Schweiz
Email: {baumann,faessler,kiser,oeztuerk,richter}@ifi.unizh.ch

Abstract: In this paper, we will talk about our experiences with implementing a first set of tools for our software maintenance project, called *AEMES*[2] which is currently being developed at the Institut für Informatik of the University of Zurich, using the functional programming language Standard ML. We will first talk about some experiences we made with this fairly new programming language and then go on to sketch the ideas that are behind our implementation of a functional COBOL parser/recognizer. We will further elaborate on why we had to introduce gross-grain scanning when dealing with COBOL programs and then talk about our cross referencer. The paper closes with conclusions and an outlook on our future work.

1 INTRODUCTION

Functional programming languages are well established today and the advantages of the functional programming approach are also widely known. The possibility of fast development, the intuitive structure of functional programs and the powerful abstractions of functional languages make the functional approach ideal for complex tasks.

In this paper, we will talk about our experiences of using Standard ML [MTH90] in developing the software maintenance environment AEMES and why we consider our approach very promising. The aim of the AEMES project is to design a prototype of a maintenance environment for COBOL 74 programs.

In the next section, we will give a brief introduction to functional programming languages and we will also talk about why we chose Standard ML as our implementation language. In the following sections, we will give some information about the parser, the program browser and the cross-referencer which we were able to implement within only a few months. We will close with an outlook on our future work.

2 STANDARD ML IN SOFTWARE MAINTENANCE

2.1 Functional Programming

The earliest programming languages were developed to provide a means of controlling the behavior of a computer. Those primitive languages were assembly languages, and attempts were soon made

[1] *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.

[2] *AEMES* stands for "**A**n **E**xtensible **M**aintenance **E**ngineering **S**ystem", and is supported by *KWF*, *Swiss Life Insurance and Pension Company* and *Bull*.

to create languages that were on a higher level of abstraction. Among the first here were FORTRAN, Algol 60 and COBOL. But the evolution of languages moved on and on a branch separate from the evolution of imperative languages, declarative languages were developed [Hud89].

The characteristic of declarative languages is that they do not have an implicit state and their emphasis is placed entirely on expressions. In the case of functional languages these expressions are composed of functions. These functions take an input item and return the value of the function application as the desired result. (Note, that this is very much in contrast to imperative programming languages, where results of 'functions' or procedures normally lay in the side-effect of a call of a function.) This behavior of functions therefore makes them particularly well suited for dealing with symbols: Symbols are taken as the input of functions, where they are combined, rearranged, manipulated and finally returned as the result of a function, all of this without destroying the input symbols or the input structure.

When thinking about doing computer supported software maintenance, one does quickly notice that a program that has to be maintained can be seen as a collection of symbols at a varying level, depending on the level of abstraction one desires: a program can be looked at as a symbol, single statements can be seen as symbols, procedures, functions, subroutines can be considered symbols, single lines of a program code can be seen as symbols or every single token of a program can be looked at as a symbol. Of course, not all of these abstractions seem to be equally useful, but their usefulness also depends very much on the task one wants to perform on a given program.

2.2 Using Standard ML

In our project AEMES we wanted to use two extremes: We choose the fairly new functional language Standard ML to create a software maintenance environment for COBOL 74 programs. Although these two languages lie far apart in their ideologies, we have been very successful in quickly developing a parser for COBOL, a program browser and a cross referencer, and we are very confident that we will succeed in developing further tools very quickly. We also found that many of the concepts we used to develop the tools mentioned for COBOL are in fact more general and could be applied to virtually any other programming language.

Standard ML stands for Standard Meta Language. However, this alone would not provide enough reason to use Standard ML as the meta language to cope with the many aspects and problems of COBOL in AEMES. Indeed, the reasons why we have chosen Standard ML are manifold.

First of all, there is the functional nature of Standard ML which allows for very fast developing of program code, since all the functions of a program can be tested separately, and be put together to larger programs afterwards. But then, Standard ML has other features that are not yet too common in (functional) programming languages:

- Standard ML has a safe static type system, which means, that types of functions are automatically deduced and checked any time functions are put together and applied. This has proven to be very helpful in writing error-free code. We also found that type mismatches very often hint at design flaws which can then be corrected at a very early stage.

- Standard ML has pattern matching in arguments of functions, so, in order to, for example, write a list processing function, the general pattern is to usually write two variants of the same function, one to catch the case of the empty list, and another one to catch the case of the non-empty list. This allows to write very readable code, avoiding many if-then-else.

- Standard ML fully supports higher-order functions. Higher-order functions are functions that have functions as their input and/or output values. This is possible since Standard ML treats functions as first class objects like any other datatype. This is a very useful mechanism to create very powerful abstractions in programs, as we will see in the next section of this paper.

3 FUNCTIONAL PARSING

This section gives a brief introduction to the principles of functional parsing. More detailed descriptions can be found in [Rea89, Pau91, Hut92]. We have also discussed our use of functional parsing in [BFKO93].

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:

```
lexer:  programs → lists of tokens
```

We will not go into the details of writing the lexer function here. It's implementation is very straight-forward.

3.1 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:

```
parser:  lists of tokens → internal reps × 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` we get:

```
parser ∘ lexer:  programs → internal reps × lists of tokens
```

3.2 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. Basic parsers are easy to implement 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.

3.3 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 EBNF-productions there exists a set of possibilities to combine terminal and non-terminal symbols to obtain new non-terminal symbols. For example, one is:

$a = b\ c$ (or as we write $a = b\ \&\ c$)

defining a in terms of b followed by c.

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 the 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 higher-order functions. Since Standard ML supports higher-order functions, the combinator functions can be expressed very easily in 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;
```

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` × `parser_type` → `parser_type`

where `parser_type` is an abbreviation for

`lists of tokens` → `internal reps` × `lists of tokens`

The other parser combinators can be defined in a similar way.

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 incremental parsing of the source code. Using this approach, we have been able to write a recognizer for COBOL 74 within only a few weeks. However, we had to modify the parser combinators slightly in order to make the recognizer for COBOL 74 fully backtrackable. We did this by always trying to find all possible parses for a program instead of only one. This has also helped us finding ambiguities in the grammar. We also used a special translator function to create all the non-basic parsers directly from the EBNF grammar of COBOL. In this way, we were able to obtain large parts of the recognizer automatically and we could refine it by simply rewriting the EBNF productions.

4 GROSS-GRAIN SCANNING AND PARSING

In the previous section we have shown, how easy it is to realize a parser for a context-free language with Standard ML, assuming there exists a description of the appropriate syntax in (E)BNF. The syntax of many of today's languages is already defined in (E)BNF. Unfortunately this is not true for COBOL.

COBOL has some unpleasant characteristics which complicate the formal treatment of this language significantly. Despite the fact that COBOL is a language that has been standardized by ANSI, there coexist today several stages of development (COBOL 68, COBOL 74, COBOL 85) together with many vendor specific dialects, which are varying greatly and comprising hundreds of keywords. The formal description of such a grammar becomes therefore colossal and it is practically impossible to do it in full generality because of the many variants of COBOL.

In our project, we have succeeded to derive an EBNF grammar for COBOL 74 from the ANSI standard [ANS74] and to realize a parser written in Standard ML using the method described in the previous section. But we had to learn, that no COBOL program running in the industry - although declared as being written in COBOL 74 - can actually be parsed without problems. This, not because our parser would not work properly, but because vendor specific variants of COBOL 74 deviate so much from the standardization document mentioned above.

To cope with this problem, we chose an approach, whose aim is not to implement a parser which is able to syntactically analyze a program within a single step. Rather, we follow a hierarchical solution, which first cuts a COBOL program into its more general structural elements, which are *divisions*, *sections*, and *paragraphs* and uses refined parsing techniques on these structural elements only when really necessary.

When comparing different variants of COBOL - now focusing on COBOL 74 - it is remarkable that, despite all the differences, there are syntactical elements that are common to all variants: Each COBOL program is, at the highest level, always organized in divisions. These divisions then are either split down into sections or paragraphs. A section may also contain paragraphs. Paragraphs are made up of a number of sentences, where a single sentence is a sequence of statements, which has to be terminated by a full-stop. Thus all COBOL 74 programs, no matter what dialect, are structured according to the following hierarchies:

`COBOL-PROGRAM` → `DIVISION` → `SECTION` → `PARAGRAPH` → `SENTENCE` → `STATEMENT`

or

`COBOL-PROGRAM` → `DIVISION` → `PARAGRAPH` → `SENTENCE` → `STATEMENT`

In practice we found, that COBOL programs can indeed be separated into these parts, using relatively simple rules. We also learned, that there is no need for sophisticated parsers to achieve this. A simple analyzing mechanism that only searches for certain patterns is already sufficient and accordingly, quite fast.

In order to achieve this dismantling one has to make use of some peculiarities of COBOL. COBOL is not a format-free, but a line-oriented language, where a line is divided into five areas which have a distinctive syntactic meaning. To dismantle a program into the three syntactical units divisions, sections, and paragraphs, it is sufficient to look at the A-area, which covers columns 8 to 11 of a COBOL line. If in a syntactically correct COBOL-Program the A-area is used, one can be sure that this indicates the beginning of a syntactical unit. Also the ending of a syntactical unit can be detected very easily. Simply, the current one ends, where the next one begins or at the end of a program. To further split a program into sentences and statements is a bit more difficult, but there is still no need for a full parser there.

The advantage of our approach lies in its independence from different COBOL variants. It is only at the level of the statement where the vendor-specific features become effective. But statements cannot be analyzed by such simple and general rules anyway. Here, the incremental nature of our COBOL parser (remember, each production of the grammar is a separate parser) is of great help. This parser allows us to analyze arbitrarily any syntactical unit of a program independently and we do not have to parse the whole program.

Another advantage of our hierarchical approach lies in the efficiency and the economic usage of memory. Since, in a maintenance-environment, not all the tools depend on a fully parsed representation of a program, it is not very useful, to translate these programs into huge, internal representations anyway.

The simple rules we have just described can without much difficulty be expressed as Standard ML functions. Mainly, the possibility of this language for symbol manipulation and pattern matching have lead to a very concise and very readable source code. Our first prototype consists of only about two dozens Standard ML functions. These functions, extended by a graphical user interface that has also been written in Standard ML, are already used in our project as simple program browser.

5 INTERACTIVE CROSS REFERENCER

A cross referencer is a program that generates a table, which contains each identifier (symbol) of a program together with references to each position in the program where the symbol occurs. In our approach, each entry in this table is a quadruple (see Fig. 1). The first entry in the tuple denotes

Class	Name	Definition	Occurrences
`Var`	i	34	34, 46, ...
`Proc`	perm	124	124, 200, ...
`Var`	a	23	46, 87, ...
`Key`	ELSE	none	46, 76, ...
⋮	⋮	⋮	⋮

Figure 1: an example of a crossreference table

the class to which a symbol belongs to. The second entry contains the name of the symbol. The third entry points to where in the program the symbol has been defined. Finally, the last entry in the tuple is a list of line numbers indicating the lines where in the program the symbol appears.

The position of the definition of a symbol in the table has been separated from the occurrence of that symbol in order to make this information more easily available. If necessary the table can easily be extended by one or more columns in order to extract more detailed information from a program text.

A cross reference table is a simple, but very important aid in the maintaining and re-engineering phase. It supports and eases the process of finding errors in a program.

In order to work quickly and effectively with the cross reference table, one has to be able to use it interactively. To allow this, we need to define a set of access functions that allow simple queries on the cross reference table. The results of these access functions can then be further combined by higher-order functions to gain more specific information. Here are some examples of functions dealing with the cross reference table (they will described later on):

- *create_crosstab*: creates and returns the cross reference table
- *get_list*: returns the list of all the names of references for a specific class
- *get_def*: returns the list of line numbers, where a particular symbol is defined
- *get_refs*: returns the list of line numbers, where a specific symbol appears

Because the cross reference table can be used interactively, one can also put a graphical user interface on top of it in order to present the information to an end user. Of course, when using a graphical user interface the access functions to the cross reference table have to be efficient enough to make working with the cross referencer agreeable for the end user.

One of the interface functions that we already have seen above will be described in more detail. The other ones can be developed in a similar way.

- *get_refs*:
  The input for this function is the cross reference table, a class and a symbol name. The output will be the list of line numbers the symbol appears in. The procedure is:
  1. Filter all the quadruples from the table that belong to the desired class.
  2. From the resulting list, filter all quadruples that have the desired symbol name.
  3. Return the 'occurrences' field.

What became clear to us was that operations like "filter all the elements that fulfill a certain predicate" or "apply a function to each element of a set" are very often used and therefore play an important role in the access functions of the cross referencer. Such operations can very easily be implemented in Standard ML as higher-order functions. In fact, operations like the ones just mentioned are so heavily used, that they are already offered in libraries to Standard ML. Since Standard ML is polymorphicaly typed, these operations can be applied to virtually any datatype. Let's look at some examples of such higher-order functions:

- *map*: `map f list`
  Signature: ('a → 'b) → 'a list → 'b list
  The higher-order function *map* takes a function f and a list as its input. *map* transforms each element of the list by applying f to each element and returns a new list of the transformed elements.

- *filter*: `filter f list`
  Signature: ('a → bool) → 'a list → 'a list
  The higher-order function *filter* takes a function (predicate) f and a list as its input. *filter* applies f to each element of the list and only returns the elements for which f is true.

With the aid of such higher-order functions the functions dealing with the cross reference table can be implemented very easily and elegantly. As an example we give *get_refs*:

- *get_refs*:

```
fun classX tuple = (* is tuple of class X ? *)
fun symbY tuple = (* is name in tuple equal to Y ? *)
fun take_ref item = (* take the fourth entry of tuple (occurrences) *)
val tup_list = filter classX crosstab
val list = filter symbY tup_list
val reflist = map take_ref list
```

The cross referencer is not a tool of outstanding complexity, even for COBOL, and can in fact be implemented in any other programming language, too. But what we want to show here as an example is that it is very easy to define and solve problems like building a cross referencer in Standard ML. By using higher-order functions, one has general patterns available that can be easily adapted to a specific problem. Higher-order functions are also very easy to combine, what makes their use even simpler. The experience that we made is that the design of a solution and its implementation go hand in hand. So the design is in the foreground, and not the implementation: "One thinks of a solution and it is halfway implemented".

6 CONCLUSIONS AND FUTURE WORK

We have started working with Standard ML four months ago, with none of us being familiar with functional programming and we now have a functional parser, the gross-grain scanner, a program browser, and a cross referencer for COBOL 74 programs. We are really satisfied with how fast one can develop tools using Standard ML and we are looking forward to our future work.

Our future plans are to develop a framework in Standard ML for control flow analysis and data flow analysis based on abstract interpretation, as well as a framework for data management of AEMES. We are also thinking about a functional way of creating a graphical user interface and integrating it into our environment.

References

[ANS74] ANSI. *Programming Language COBOL.* American National Standard Institute, New York, X3.23-1974 edition, May 1974. ISO 1989-1978.

[BFKO93] Peter Baumann, Juerg Faessler, Markus Kiser, and Zafer Oeztuerk. Using functional parsing to achieve quality in software maintenance. *Submitted to Second International Conference on Achieving Quality in Software*, October 1993. Venice.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

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

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

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