

Syntactic Parsing in FSharp Using Monadic Parsers

Jarno Holstein

supervised by
Dr. Giuseppe Maggiore

Abstract

The previous compiler for the Meta-Casanova language became obsolete when a new version of the language was created. Updating the previous compiler to the current version of the Meta-Casanova language would have resulted in unreadable and unreliable code. In order to ensure correctness of code and ability for future maintenance, a new compiler will be developed. The new compiler will allow for faster maintenance and easy modification resulting in less time wasted and less bugs in the software.

Contents

1	Introduction	1
2	Background	2
3	The plan	3
4	Details	4
5	References	8
6	Conclusion	8

1 Introduction

There are a lot of programming languages on this planet. Most of them have a working compiler. So it seems that designing a compiler is a problem that has been long solved. There are even programs that generate the parts needed to create a front end for a compiler (Flex, Bison) and programs that function as entire backend's (LLVM). So why, if we have access to tools that simplify the creating progress, build a compiler from scratch?

Because you want or need your compiler to do something extra that those tools are not able to provide. Or at least not in a way that would negate the reason you wanted to use those tools in the first place: simplicity of code.

You want a compiler that gives useful feedback about the code it compiles. You want a compiler that is easy to maintain or modify. You want a compiler that processes code exactly like you want it. You want a compiler from scratch.

This paper is part of a collection of three papers. The subject of this paper is monadic parsing in the front-end. For information on the Meta-Casanova language and its standard library, refer to the paper by Louis van der Burg [1]. For information about the extended type system of Meta-Casanova, refer to the paper by Douwe van Gijn [2].

2 Background

Meta-Casanova is a language still in development [1]. Its syntax changes and functionality is added over time. As Meta-Casanova evolves further, the compiler for said language must also be able to cope with the changes made. This however is where we ran into a problem. When a new and stronger type system was added to the language [2], the old Mark Two compiler could not simply be adapted to incorporate the changes. If it were a change in the syntax only a few changes in the code would have been needed to achieve the desired goal as the structure of the language would have stayed the same. However in order to implement the stronger type system in the mark two compiler, there was code needed to be added to a sys-

tem that was not designed for such expansion.

In the end it was decided that creating a new compiler from scratch would produce a greater reward than adding a new structure to the older compiler. The greater reward being that it would provide the chance to completely change the structure of how the compiler works, allowing us to implement a structure that would be easy to read and adjust when future changes are required.

When the maintainability of the code becomes less of a chore we can focus our effort into code correctness. And seeing that a compiler above all needs to be correct gives us the motivation to start developing our new compiler, the Mark Three.

3 The plan

Basic idea of the front-end

First we should look at the basic design of the front-end. The front-end of our compiler consist of an lexer, parser and type-checker. The lexer should be able to indentify keywords, convert a list of digits into a integer or floating-point number and convert a list of characters into a string if the characters are enclosed by the " symbol. Once lexing is completed there should be a list of tokens as result.

The tokens need to be detailed enough for the parser so it never has to manipulate the data they contain, only read it. We do this in a way that the parser won't have to do the tedious job of indentifying, what should be, the smallest components in the language. This makes the parser code easier to read and makes compiling faster. Consider that the list of characters given as input always needs to be identified before every syntactic check, if the parser needs to do this identification before very check you would end up with an O notation of $O(n^2)$ for the parser. However if the lexer indentifies all tokens before parsing you will get $O(n)$ for the lexer and $O(n)$ for the parser. Meaning a $O(n)$ for the entire lexing parsing process.

The parser needs to build a data tree from a list of tokens. It also needs to be able to find errors in the syntax and give useful feedback on these errors such as the position of where the error occurred and possibly what kind of error. This data tree often coincides with the parse

tree of the parser(see below in details).

The type checker does as the name implies, checking types. Once parsing is done there should be a data tree that the type checker can traverse. In this data tree all connections between the components of the language can be found. It is the type checkers job to find if these connections are sound and if this is not the case then give useful error messages as feedback. This process generates type information that will be used in the back-end of the compiler during optimization and code generation.

The idea

Now that we have a basic concept of the front-end we can start looking into the structures needed to keep the code readable and maintainable.

The key to maintainable code is writing short code. If there is less code, there is less to maintain. In order to achieve short code we must decide upon a structure that allows code to be repeated. If we create code that can be used as building blocks, we create an environment where we can quickly and correctly create new structures that are easy to maintain in the future.

A "building block" that is used in the lexer and parser is the Parser Monad (See Parser Monad in chapter 4). The Parser Monad in combination with Parser Combinators allow us a higher form of programming. A problems such as error handling is already integrated into the monad making it easy to implement.

4 Details

In this chapter we will go through several subjects that have been handled above in more detail,

Why not use a Flex generated lexer?

Although Flex generates powerful lexers it forces you to work in a certain way that makes it difficult to add things such as line and column detection or error support. Furthermore the frontend discussed in this paper is written in a functional language (FSharp) and thus even though you could incorporate a Flex generated lexer in FSharp it would be safer and more future proof to make your own lexer in FSharp.

Why not use a Bison generated parser?

For the same reasons why we are not using Flex. There are also certain features of the language we are parsing that would be more difficult to implement in Bison. For instance take the next code example.

```
Func """add -> int -> int -> int
add a b -> a + b
```

Listing 1: simple add example.

In this example we have a relatively simple structure. First we declare the function add that will take two int's and then return an int. Then we define the function add and bind the variable a to the first int and variable b to the second int. Finally we return the sum of a and b. This example is easily parsed by Bison because it is predictable. It can also easily be parsed by a Bottom-up structure. However take in the next example.

```
Func int -> """sub -> int -> int
a sub b -> a - b
```

Listing 2: more complex add example.

What happened here is that we in-

cluded the ability to bind variables on the left side of the function. Note that the two examples above do not exclude each other. Both could exist in the same file. Bison has trouble parsing this situation because it isn't sure what to call an variable and what to call a function-name. If you could guarantee that there will always be one variable on the left side of a function, Bison would be able to parse it because it would be easy to predict in a Bottom-up structure. However in the nature of this language that is not something we can know for sure and thus bison has trouble parsing this.

What about the type checker?

In order to compile code we need to know all the types of the functions variables and literals that are present. For instance:

```
Func """foo -> int -> bool
```

Listing 3: static declaration.

In this declaration we know all the types of function foo. We know it will take an int and return a bool. This static way of declaring types is not the only way to do so.

```
Func """foo -> 'a-> 'a
Func """bar -> int -> int
Func """vla -> bool -> bool
foo x -> x
bar y -> foo y
vla z -> foo z
```

Listing 4: dynamic declaration.

Here we have a dynamically typed declaration. In this declaration of foo we don't have a clear insight as to what the types are of the arguments and the return value. The only way to determine what the types are is to look at the definitions. In the definition of foo we don't have indications what the types could be. However if we look further we can see that the function bar uses foo and bar

does have clear types. So if we cross-reference the types between bar and foo we can conclude that foo takes an int and returns an int.

But if we look at the definition of vla can see that vla also uses foo and this time foo takes a bool and returns a bool. This is the power of dynamic typeing. We can give a generic declaration and use it to make multiple definitions of the same function using different types.

Of course this only works if the definition of foo is equally generic.

```
foo x -> x + 5
```

Listing 5: Added foo.

If we make foo adds five to the end result we make foo less generic. We now have limited the optionsfoo can have as types to the types that allow five to be added to them.

Parse Tree

The parse tree is an representation of all the structures in a language. It defines all the correct syntaxthat the language is capable of. Let's give an simple example of a parse tree.

```
(Lines)
  (Line) (Lines)
  (Line)
(Line)
  (Define)
  (Print)
  Newline
(Define)
  <Define> Name (Content)
(Content)
  Value (Content)
  Operator (Content)
  Function (Content)
  Newline
(Print)
  <Print> Name Newline
```

Listing 6: Parser Tree Example.

Here we have a outline of an simple programming language where we can define and print data. The code that would fit this tree would look like this:

```
Define foo 5 + 4
Define bar foo + 1
Print bar
```

Listing 7: simple language example.

We will now analyze the example code and parse tree in further detail.

As you can see the language has two keywords namely Define and Print. When the keyword Define is found, the parser will expect an name and content to be defined. The Name will be easy to parse because it can only be one thing, in this case a string of characters. Content however can be multiple things as seen in the parse tree. If we look at the Content after the definition of foo we could parse this Content as: Value <- Operator <- Value <- Newline with 5 and 4 being the values, plus being the operator and Newline being the promise that the code will continue on the next line. The definition of bar has a similar Content namely: function <- operator <- value <- newline.

Lines are made up out of either a line followed with another line or just a singular line. A line consists of either a definition or a print statement. Note that a line can also consist of a newline. If newline is not specified in the parse tree than empty lines in the code aren't correct syntax and will result in an error. Following this logic you may have noticed that if there isn't a newline after the last line of code then the content wouldn't parse and neither would the print statement because they require a newline to terminate.

Parser Monad

The signature of the parser monad used in our parser is :

```
type Parser<'char,'ctxt,'result> =
  List<'char> * 'ctxt -> Result<'char,'
  ctxt,'result>
```

Listing 8: Parser Monad signature.

As you can see it takes a list of a certain type and a context that could be any data structure you wanted it to be and

then returns it as a Result.

The signature of result is as followed:

```
type Result<'char,'ctxt,'result> =  
  | Done of 'result * List<'char> * 'ctxt  
  | Error of ErrorType
```

Listing 9: Result type.

Result can be a Done of an result, a list of a certain type and a context. The Done represents the output of a parser monad that has successfully completed its task. Error on the other hand represents a monad that has failed. Let's take an example from the lexer to show how the parser monad would operate:

```
let char (expected:char) :Parser<char,  
  Position,Unit> =  
pr{  
  let! next_char = step  
  if next_char = expected then  
    do! increment_col  
    return ()  
  else  
    let! ctxt = getContext  
    do! fail (LexerError ctxt)  
  }  
}
```

Listing 10: Char Parser.

Here we see a parser monad that checks if the next char in the list is equal to the char that is expected. Step is also an monad that returns the next element in a list. As you can see an simple if-statement is used to compare the next char to the expected char. If the statement is successful the position will be modified by incrementing the column number because we moved a column to the right. Position in this case is the context of the monad. If the statement fails however we will return an LexerError containing the position information. Now we have a parser monad that can check a single char and are able to create more complex parsers like:

```
let horizontal_bar pos :Parser<char,  
  Position,Token> =  
  pr{  
    do! char '-'  
    do! char '-'  
    do! char '-' |> repeat |> ignore  
    return Keyword(HorizontalBar,pos)  
  }
```

Listing 11: Bar Parser.

Here we have a parser monad that succeeds when two or more dashes are detected. If one of these char monads fail then the horizontal-bar monad will fail also.

Parser combinators

The main reason why we use monadic parsers is for the use of parser combinators. A parser combinator is a monadic parser itself, one that takes either one or more parser monads and changes the given monads to give a desired effect. Let's look at a few examples.

```
let (.>) (l) (r) =  
  pr{  
    let! res = P1  
    do! P2  
    return res  
  }
```

```
char '('a) .>> char '('b)
```

Listing 12: .> example.

Here we see the basic parser operator (.>). This operator will succeed when both of the parser monads given succeed. However upon success the (.>) operator will only return the result of the left parser given. There are other variants of this operator that do a similar function such as (>.) which returns the result of the right parser given if both parsers succeed. And (>.) which returns both results in tuple form if both parsers succeed.

Another basic operator that gets a lot of use is the (.||) operator.


```

let inline (.||) (p1:Parser<_,_, 'a>) (
    p2:Parser<_,_, 'a>) : Parser<_,_, 'a>
    =
    fun (chars, ctxt) ->
        match p1(chars, ctxt) with
        | Error(p1) ->
            match p2(chars, ctxt) with
            | Error(p2) -> Error(p2)
            | Done(res2, chars', ctxt') ->
                (res2, chars', ctxt') |> Done
        | Done(res1, chars', ctxt') ->
            (res1, chars', ctxt') |> Done

```

Listing 13: .|| example.

Similar to the (.») operator it takes two parsers however where the (.») has to execute both parsers given the (.||) only needs to execute one at best. If the left parser succeeds the right parser won't be executed. However if the left parser fails the right parser gets executed. So unlike (.») the result upon success depends on which parser succeeds first. In case both parsers given fail the error given by the last parser executed will be the error returned by the operator.

This time the dot symbolizes the parser that is first executed. With these operators we now have access to simple logic which we can use to create more complex parser combinators.

```

let rec repeat (p) =
    prs{
        let! x = p
        let! xs = repeat p
        return x :: xs
    } .|| prs {return []}

```

```
repeat char '-'
```

Listing 14: repeat example.

The repeat combinator takes a parser and transforms it into a parser that repeats the given parser until the given parser fails. In the case of "repeat char ('-')" the parser created will take "-" from the character list until it encounters something that is not a "-". You may wonder what the result of such a parser may be. Well, seeing that char('-') returns unit as result repeat char ('-') will return a list of units.

A problem with repeat is that it will never give an error as it uses the error to identify the end of the sequence. So if repeat "fails" the result will be an empty list. There are situations where we want to preserve the error messages that repeat comes across. So we will look at iterate:

```

let rec iterate (p:Parser<_,_, 'result>)
    : Parser<_,_, List<'result>> =
    prs{return! eof >>. ret []} .||
    prs{
        let! x = p
        let! xs = iterate p
        return x::xs
    }

```

Listing 15: iterate example.

Iterate will succeed if all of the elements from the starting position to the end of file succeed. If iterate fails the last element that failed will give the error message returned by iterate.

5 References

- [1] Burg, Louis van der. Adventures in Meta-Casanova. 2016
- [2] Gijn, Douwe van. Extending Meta-Casanova with Modules and Type Operators. 2016

6 Conclusion

In this paper we looked at structures that we could find in a monadic parser. We have discussed the benefits that such a structure could provide as have we gone in detail what such structures could entail.