Design and implementation of the Meta Casanova 3 Compiler front-end

Bachelor thesis

Jarno Holstein
0860836

Hogeschool Rotterdam

June 12, 2016

Abstract

This document describes the development of the front-end and type checker of the bootstrap compiler for the Meta Casanova 3 programming language. It is written as a bachelor thesis for the Rotterdam University of applied sciences.

The front-end and type checker are responsible for parsing and analyzing Meta Casanova mark 3 source code and generating a data structure that the back-end can use to create an executable. In this thesis we will show how how the front-end and type checker function and how their design came to be.

Contents

1 Intro

Creating games often proves challenging from both a design perspective and a programming perspective []. In order to simplify the problems at the programming side of this challenge, a new programming language was created. The programming language (Casanova) is designed to better express the abstractions found in games. During development of Casanova the research team found that the code of the Casanova compiler became difficult to maintain. To address this issue, a new language was created: Meta Casanova.

1.1 Context

1.1.1 The company

The graduation assignment is be carried out at Kenniscentrum Creating 010. The company is located in Rotterdam. Kenniscentrum Creating 010 is a transdisciplinary design-inclusive Research Center enabling citizens, students and creative industry making the future of Rotterdam[].
The assignment is carried out within a research group, who is building a new programming language. The new programming language is called Casanova.

1.1.2 The research group

The research group that focuses on the Casanova Language comprises of the following members: Francesco di Giacomo, Mohamed Abbadi, Agostino Cortesi, Giuseppe Maggiore and Pieter Spronck.
In the research group there is a team that works on developing Meta Casanova and its compiler. This team consists of interns, namely:

Jarno Holstein, that works on the front-end and back-end of the compiler.
Louis van der Burg, who develops and debugs the MC language.
And Douwe van Gijn, who develops the back-end of the compiler.

1.1.3 Motive

Meta Casanova is a programming language meant for the development of compilers. The need for such a language arose after the research group for the Casanova programming language found that their compiler became difficult to maintain and adapt. This difficulty originated from the fact that F#, the language Casanova was written in, did not support the higher level of abstractions that compiler developers could benefit from. Thus MC was created. MC proved itself preferable over F# when the Casanova compiler was rewritten in MC. What used to be 1480 lines of code in F# are now 300 lines in MC [].
The MC compiler does however have a few drawbacks. The compiler does not give useful error messages, if something is wrong in the MC code it will be hard to identify exactly what. It is difficult to maintain and expand the compilers code. So when a new and improved version of MC was developed, namely MC mark 3, the research team decided to also fix the drawbacks that the compiler was facing by developing a new compiler.

2 Research questions

The goal for this project is to create a maintainable and extendable front-end and type checker that future developers can continue with. The following research question could be formed of this goal. How to develop a maintainable and extendable front-end and type-checker for the MC mark 3 programming language? In order to answer the main question the following sub question were formed. What properties does MC have that the front-end needs to process? How to develop a maintainable and extendable parser for MC? How to apply a type-system to MC? The project comes with the following requirements:

2.1 Adaptability requirement

The code created during this project should be easily maintainable by developers who did not initially design the structure of the compiler.

2.2 Correctness requirement

The program is a compiler and has an even smaller margin for bugs than other programs have. No program the compiler compiles would be bug-free if the compiler itself is not bug-free.

2.3 Diagnostic requirement

The front-end and type checker need to be able to give descriptive error information for the programmer of MC.

3 Meta Casanova

Before we can show the compiler, we need to have a look at the language that it needs to compile. In this chapter we will explore the syntax and features of Meta Casanova mark 3.

3.1 Functions

The functions in MC first need to be declared before they can be defined in rules. We do so like this.

```
Func ""foo → int → int → int
```

In the example above we can see a function declaration of the function "foo". It consists of the keyword "Func" followed by a function name and type signature. The type signature is a sequence of types that the function can take as input. The types in this sequence is separated by the arrow keyword "→" and ends with the return type of the function.
The type signature can also be declared in a way that would allow the function name to be infix:

```
Func int → ""bar → int → int  > L 5odete the priority arrow
    ""> that is added to the declaration.
is priority arrow allows the programmer to assign priority and associativity to the
    function.
inebreak
 define the functions we declared we need to add rules.
inebreak
 the next example we can see both ""foo and ""bar defined as single rules.
cause of the simplicity of the rule we can define it without using premises.

egin{code}
      foo a b → a + b
      a bar b → a + b
```

It can however be defined with premises as shown below.

```
a + b → res
--------
foo a b → res
```

As shown, the premise is located above the input output line of the rule and separated with a horizontal line.
It is possible to define multiple rules for the same function.

```
a < 5
a + b → res
---------
foo a b → res

a >= 5
a - b → res
---------
foo a b → res
```

Here we have a situation in which the function foo has multiple rules. When such a situation arises the rules will be executed in the same order as they are found in its file when the function is called. As you can see the first premise of both rules has a different kind if premise in it, a conditional. If value "a" is less then 5 then the first rule will succeed the second rule will be ignored. And if "a" is equal or

greater then 5 the first rule will not match and the the second rule will succeed. If both rules wont the program will crash.

3.2 Data

Data declarations are used to create data structures. For example, to declare a tuple of two int's we write:

```
Data int → "", → int → ituple
```

Above we see a data declaration. It consists of the keyword "Data" followed by a name and a type signature. The data structure "," is capable of containing two int's in a container with the type "ituple". Below we see an example of how this tuple can be used.

```
Func "give-back-"tuple → int → int → ituple
Func "give-back-"left → ituple → int
Func "give-back-"right → ituple → int

a,b → res
-----
give-back-tuple a b → res

a → left,nope
-----
give-back-left a → left

a → nope,right
-----
give-back-right a → right
```

Here we can see the two things that Data in MC can do, namely constructing and deconstucting data structures. When data is used on the left side of the premise arrow it takes arguments and stores them into the data structure. And when data is used on the right side of the premise arrow it will take the data structure and binds it content to new value names.
Unions of data can also be made by using the same return type.

```
Data ""F → float → fiunion
Data ""I → int → fiunion

a → F contentA
b → F contentB
F (contentA + contentB) → res
-------
foo a b → res

a → I contentA
b → I contentB
I (contentA + contentB) → res
-------
foo a b → res
```

The float and the int can both be contained by the type "fiunion" because of the "F" and "I" data deconstructors that share the same type, namely "fiunion". These rule definitions of "foo" can be shortened to the following:

```
foo (F contentA) (F contentb) → F (contentA + contentB)
foo (I contentA) (I contentb) → I (contentA + contentB)
```

The matching of data deconstructors "F" and "I" are now done in the input space

of the rule definition.

3.3 Generic declaration

A generic declaration allows for a function to become polymorphic with its types. This means that one declaration can take to multiple type signatures. Information must be added to the declarations to make functions and data generic. We do this by marking type vars with an " ' ".

```
Func ""foo → `a → `a
```

The " ' " in front of the type names denotes a type variable. These generic types are used to give the type-checker information during type inference. For instance

```
Func ""bar → int
bar → foo 5
```

We have the function "bar" that returns the result of function "foo". In the declaration of "bar" it is specified that it will return an int. In the declaration of "foo" it is specified that the type of the first argument is the same as its return type. "foo" is given a 5 which is a int. Following these facts we can deduce that everything type checks. We can do this without knowing the definition of "foo" because the declaration holds all the type information we need.
Here we have "foo" take a different type as its input, namely a float.

```
bar → foo 5.5
```

In the example above we run into a type error because "foo" will now return a float which conflicts with the int that "bar" is expecting.
There are situations where the type information in the declaration wont be enough to inference all the types. For example:

```
Func ""bas → `a → `b
```

In the function "bas" we now have two generic types that are independent to each other. The only way to find their relation to one another is to look in the definition of "bas" and see how they are connected.

3.4 Polymorphic data structures

In data, generic arguments have additional consequences. Because data structures need to be able to construct to a container and deconstruct from a container, additional type information is needed in the return type of the data declaration. Take in the following code.

```
Func ""foo → tuple
Data `a → "", → int → tuple
foo → 5 , 5
foo → 5.5 ,  5
```

In the code above we have function "foo" that returns a tuple. In one of the foo's we return a "," with two int's and in the other we return a "," of a float and an int. They are both tuple according to "," however we have no way to verify this in the syntax. What if the definition of foo doesn't work with a "," that carries two int's? Normally we would specify in the declaration the exact type that foo requires but how do we do that when generics are involved?
Our solution was special type-application syntax.

```
Func ""foo → tuple<float>
Data `a → "", → int → tuple'<a>
foo → 5.5 ,  5
```

We added a way to specify sub-types in data structures. This way we have full control of what types a function needs. It also gives us a way of abstraction. For example, we can define an universal tuple.

```
Data `a → "", → `b → tuple<`a `b>
```

This gives us a universal tuple that can be re-used to contain all sorts of types.

3.5 Type functions

In order to show all the functionality of a type function we first need to know about the three levels in which MC variables can operate.

-Terms
-Types
-Kinds

Lets first look at the one we are most familiar with at the moment, the term. In short, terms carry values. Below we can see an example of this.

```
a + 5 → b
int-float-add b 5.5 → res
------------
foo a → res
```

If we look at the function definition above we see an example of simple value manipulation. The value holder "b" equals value holder "a" plus five. Value holder "res" equals value holder "b" plus five point five.
All the value holders are terms and are able to execute on run-time because their type information is known.
Types on the other hand need to be executed on compile-time in order to get all the type information for the compiled program.
In the function declaration below we see a function with two types. These types are constant and can not be changed by the compiler.

```
Func ""foo → int → float
```

However if we use a generic like so:

```
Func ""bar → `a → `a
```

the types become variable and are now dependent on the function that calls "bar".
You can already see the relation between terms and types.

```
Func ""vas → int
vas → 5
```

A term gets its identity from a type. For instance, 5 is a int, 5.5 is a float and "str" is a string.
Kind have a similar relation with types. Kinds are to types as types are to terms. In order to explain this we will now look at type functions.
The example below shows both familiar syntax and new syntax. We can still recognize that the code above consists of a declaration and definition. what is new though are the keywords "Typefunc", "Type" and "=>".

```
TypeFunc Type => “”* => Type => Type
a * b => tuple<a b>
```

In the declaration we see that the keyword “TypeFunc” is used to start the declaration of a type function. Notice that the single arrow “→” that is normally used in declarations is replaced by a double arrow “=>”. The deference between these arrows signify if the code is supposed to be run-time or compile-time. And lastly we see the keyword “Type” which is used to tell that the type function “*” will take two types and return a type.
In short the keyword “Type” signifies the use of a kind.
Lets give a example of when this is useful. Remember back above with the generic data tuple:

```
Data `a → “”, → `b → tuple<`a `b>
```

we need to specify what the sub types of tuple are in order to make tuple generic. Now every time you want to use this generic tuple you need to give the sub types to tuple.

```
Func “”foo → int → int → tuple<int int>
```

This is still do able but when you are using more complex types like:

```
Func “”bar → tuple<int int> → tuple<tuple<int int> int> → tuple<tuple<int int
> tuple<tuple<int int> int>>
```

things become tedious. Type functions offer a solution for this problem. For instance, if we use the type function “*” specified above:

```
Func “”bar int * int → (int * int ) * int → (int * int) * ((int * int) * int)
```

the declaration becomes more manageable.
As you can see we are using type functions in a declaration. It is because type functions are done on compile-time that they can manipulate type information. Aside of manipulating type information type functions also have all the capabilities that a normal function has. Type functions are able to manipulate all the levels of MC, namely terms, types and kinds. As example:

```
TypeFunc “”add => int => int => int
add a b => a + b
```

not only works perfectly fine but also gets done on compile time. The ability to decide which code sould run on compile time can be useful for optimization purposes.

3.6 Type aliases

Type aliases are to Data as type functions are to functions. Type alias is on compile-time and is capable of manipulating terms, types and kinds. So just like you can make a tuple with data, you can also make a tuple with a type alias. Like so:

```
TypeAlias Type => “ali-”tuple => Type => Type
a ali-tuple b => alituple<a b>
```

The difference between this tuple and the tuple we declared with data is that the tuple declared with an alias works on all levels of MC: terms, types and kinds.

3.7 Modules

And lastly we look at modules. Modules are created by a type function and can contain all the aforementioned syntax. Below we can see an example of a module being declared and defined.

```
TypeFunc “”mod => Module
Mod => Module {
        Data “int-”container → int → intcontainer
        Func “add-int-”container → intcontainer → int → intcontainer
        a + b = c
        int-container c → res
        -----
        add-int-container (int-container a) b → res
}
```

Here we can see a module “mod” being created. The functions and data inside a module can be called outside a module by use of the carret.

```
Foo a b → add-int-container^mod a b
```

As you can see the syntax is slightly different than you would expect. Instead of it being a namespace followed by a function name it is the other way around, a name and than a namespace.

4 Reasearch

4.1 Overview of the front-end

In this chapter we will first take a global look at the front-end and type checker of the compiler. Then we look at examples of why tools for generating a front-end were not chosen for creating the MetaCasanova compiler.
First we look at the basic design of the front-end. The front-end of our compiler consist of a lexer and parser. The lexer is able to identify 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. The lexing process produces a list of tokens as result.
The tokens need to be detailed enough for the parser so the parser never has to manipulate the data they contain, only read it. This is done in a way so that the parser will not have to do the tedious job of identifying 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 to the parser always needs to be identified before every syntactic check, if the parser needs to do this identification before every check the parser would end up with a complexity of O(n2). However if the lexer identifies all tokens before parsing the lexer will get O(n) and O(n) for the parser. Meaning a O(n) for the entire lexing parsing process.
The parser fills a data structure from the list of tokens provided by the lexer. This data structure closely resembles the syntax tree that specifies the syntax of the language that needs to be parsed. The parser also needs to be able to find errors in the syntax and give useful feedback about these errors such as the position of where the error occurred and the kind of error.
The type checker uses a type system to check if the functions and variables in a program are compatible with each other. A type system allows a function or variable to be identified as a certain type. Once types are identified, rules can be applied to the types to check for validity of the code on compile time.
Once parsing is done there is 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 verify that these connections are sound and if this is not the case, 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.

4.1.1 Overview abstractions

In the previous section we showed the basic components of a front-end and mid-end of a compiler. Now 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 shorten code we must use a structure that allows code to be re-used. If we create code with a modular structure, we create an environment where we can quickly and correctly create new structures that are easy to maintain in the future.
In order to achieve this for the lexer and parser we use the the Parser Monad. The Parser Monad in combination with Parser Combinators allow us a higher level of abstraction. A feature such as error handling is already integrated into the parser

monad making it easy to implement.

4.1.2 Why we do not use lexer generators

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. The reason these difficulties arise is because Flex is a lexer generator. Because of its generating nature it leaves little room for customization. Furthermore the front-end discussed in this paper is written in an functional language (F#) and even though you could incorporate a Flex generated lexer in F# it would be safer and more future proof to made your own lexer in F#.

4.1.3 Why we did not use a Yacc generated parser

For the same reasons we are not using Flex. There are also certain features of the language we are parsing that would be more difficult to implement in Yacc. The next example is easy to parse:

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

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 Yacc because it is predictable. It can also easily be parsed by a Bottom-up structure. The next example is not simply parsed.

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

In this example we added an argument at the left side of the function name. Note that the two examples above do not exclude each other. Both could exist in the same file. Yacc 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, Yacc 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 Yacc has trouble parsing this. If the function was build into the parser then it would work seeing as it would have a static name to find. However in the case of sub the function name was declared in the code itself and thus would require you to first parse the code to acquire the function name and then to generating a parser that could identify the definition of the function.

4.1.4 Basic type checking

In order to compile code we need to know all the types of the functions variables and literals that are present. For instance there will be cases were the declaration knows all the types:

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

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. Below we have a dynamically typed declaration.

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

In this declaration of foo we do not 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 do not 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” we 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. For instance:

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

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

4.2 Concepts of parsing

In the previous chapter we gave a global overview op parsing. In this chapter we will show the structures and abstractions used in the compiler.

4.2.1 Parse tree

The syntax tree is the representation of all the syntax in a language. This language was made up for explaining the parse tree, it is not MC. The parse tree defines all the correct syntax that 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
```

Here we have an 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
```

We will now analyze the example code and parse tree in further detail. As you can see the language has two keywords: “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 are not correct syntax and will result in an error. Following this logic you can see that if there is not 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.

4.2.2 Monad

Before we can look into the abstractions that are used in the compiler we first need to know about monads.
Monads are interfaces to container-like objects with two generic functions. One function that allows data to be stored in the container and one function that allows for a function to be applied to the data in the container and puts the result of that function into a new container.
The return is the function that allows for a value to be stored inside the monad. Here we have some example code of a monad and its return function written in F#.

```
type Monad<t> = {I : t}
let ret (x:`a) :Monad<`a> = {I = x}
```

In the example above we see a structure Monad that can hold a value of a generic type. We also see the function “ret” that takes an value of a generic type as argument and returns a “Monad” of the same generic type.
The bind is a function that applies a function to the value inside a given monad and returns a new monad. Below we see an implementation of a bind done in f#.

```
let bind (M:Monad<`a>) (k:`a → Monad<`b>): Monad<`b> = k M.I
```

This way we can chain monads to each other like so:

```
let f1 = (fun a → {I = a + 4})
let f2 = (fun a → {I = a - 6})
let res = ret 4 |>  bind f1 |> bind f2
```

We have two functions “f1” and “f2” that take a value and return a monad that contains the result of the function. We then put the value “4” inside a monad with the return function and pass it to a bind function that takes “f1” as the function to bind the input monad to. After “f1” is bound the resulting monad is passed

to a second bind function that binds the input monad to the “f2” function. Once executed the variable “res” will contain a monad with the value “2” inside.

4.2.3 Computational statements

In F# there is syntactic sugar to make the use of a monad easier for the programmer.
To make use of this syntactic sugar we first need to create a interface for the F# compiler. Below we see such a interface for the monad that was used above,

```
type MonadBuilder() =
        member this.Bind m k = bind m k
        member this.Return x = ret x
        member this RetrunFrom x = x
let mon = MonadBuilder()
```

Now we can rewrite the bind example above using computational statements as syntactic sugar:

```
let res = ret 4 |> Mon {
        do! f1
        do! f2
}
```

Which ones again results in a monad containing the number 2.

4.2.4 Parser monad

Parser monads are monads that have the characteristics of a parser. This means that the monad takes a list of elements to traverse over and builds a data structure as a result. Because it is also a monad the parser monads can be used to form an algebra of parsers. Parser monads can thus be combined with the use of operators called parser combinators. The parser monad is a powerful abstraction that makes building complex parsers more intuitive.
The signature of the parser monad used in our parser is :

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

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. A result represents the entire result of a parser monad, be it the state of the parser monad, the return value or if the parser monad succeeded or not.
The signature of result is as followed:

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

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.
The bind of the parser monad takes a parser monad and a function to bind to that parser monad and returns a new parser monad that is the result of the function that was bound. The bind will match the parser monad that it gets as input for either a “Done” that contains a value or it will match with an “Error” that contains error information. In case that the “Done” matches the value stored in the “Done”

will be given to the function that the parser monad had reseed as input. This function will then return a new parser monad which is the result of the bind. In case the “Error” matches the “Error” that was matched will be the result of the bind. The bind function of the parser monad is as follows:

```
  let bind (p:Parser<`char,`ctxt,`result>, k:`result->Parser<`char,`ctxt,`result
>) =
          fun (chars,ctxt) ->
                  match p (chars, ctxt) with
                  | Error(e,p) → Error(e,p)
                  | Done (res,chars`,ctxt`) →
                          k res (chars`, ctxt`)
```

Bind takes the parser that needs to be executed (“p”) and checks if the it failed or if it was successful. If “p” failed than the error that “p” generated will be returned by the bind. But if “p” succeeded, then the continuation (“k”) will take the result of “p” and creates the next parser in the bind sequence with it.
The return function is simpler then the bind because it only takes a value and stores it in a parser monad. That parser monad will be the result of the return function. Below we see an example of the return function of the parser monad.

```
let ret (res:`result): Parser<`char,`ctxt,`result> =
fun (chars,ctxt) = Done(res,chars,ctxt)
```

Now that we have an understanding of how a parser can be constructed we can look at an example of a parser monad. Here we see a parser monad that checks if the next char in the list is equal to the char that is expected.

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

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.
Here we have a parser monad that succeeds when two or more dashes are detected.

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

If one of these char monads fail then the horizontal-bar monad will fail also.

4.2.5 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.
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.

```
let (.>>) (l) (r) =
        prs{
                let! res = P1
                do! P2
                return res
        }
}
char ''(a) .>> char ''(b)
```

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. 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.

```
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
```

If the left parser succeeds the right parser will not 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 7 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.
For example the repeat combinator. The repeat combinator takes a parser and transforms it into a parser that repeats the given parser until the given parser fails.

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

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 a empty list. There are situations where we want to preserve the error messages that repeat comes across. So we will look at iterate. 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.

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

4.3 Developing the parser

Now we know about what a parser does and the abstractions used in them we will look at the structure and design choices of the parser itself.

4.3.1 Modular parser

To work in iterations the parser is designed to be modular so we can easily add new functionality. It achieves this modularity by making the parser multi pass which means that it takes multiple passes to parse a single file.
The passes are separated into four groups, namely:

-Type function and Type alias declarations.
-Type function and Type alias definitions.
-Data and function declarations.
-Function definitions.

The reason that these groups are separated like this is because of how dependencies of the language works.

Function definitions need the function declarations and may need the other groups.

Data may need type functions and type aliases.

Type function and Type alias definitions need their declarations and may need other type functions and aliases.

And the Type function and Type alias declarations may need other type functions and aliases.

4.4 Developing the type checker

4.4.1 Normalizing before type-checking

If the input of the type checker is normalized the type checker becomes simpler. The normalized parser-ast is similar to the back-end interface but it also allows for generic types.

4.4.2 Modular Type checker

The type checker is separated into several modules that each check the types of a part of the MC mark 3 language . Rule type checkers Function type checker Data type checker Type alias type checker Type function type checker

5 Results

In this chapter we will look at an overview of what has been achieved during this project.

5.1 Accomplishing requirements

There were several requirement that needed to be reached so that this project would be successful.

5.1.1 Adaptability requirement

The code created during this project should be easily maintainable by developers who did not initially design the structure of the compiler.
The front-end and type checker both are of modular design making reasoning about the program structure easier to do. The parser monad provides an intuitive way of programming making the code easier to read. The coding was done following coding rules outlined in the book “clean code”[].

5.1.2 Correctness requirement

The program is a compiler and has an even smaller margin for bugs than other programs have. No program the compiler compiles would be bug-free if the compiler itself is not bug-free.
To test the correctness of the front-end and compiler several test files containing MC code were created. There files would be parsed and the output of the parser would be manually checked for correctness. There were also files that would contain several errors and the parser would need to identify what is wrong with the code. In a later stage of development, when the back-end was capable of generating code, the output from the type checker was fed to the back-end. This resulted in executable code that worked as expected.

5.1.3 Diagnostic requirement

The front-end and type checker need to be able to give descriptive error information for the programmer of MC.
A new Error system was developed for the parser monad. By giving errors priority over each other and giving the programmer a way to easily interact with the error priority, the error messages returned by the parser become more precise. It can give the exact location of the error and give the correct error message.

5.2 Other results

New knowledge of the MC programming language was discovered during the development of the front-end and type checker.

The MC syntax was tweaked to allow better parsing.

Premises were added to the declarations for easier manipulation of type information.

Information about lexers and parsers was gained and documented. Information about parser monads was gained and documented.

An easy to maintain Front-end was created that could parse MC3 entirely and a modular type checker was created that could check MC2 with dot net calls.

A minimalistic parser for MC3 syntax was created at request of a costumer that had need for it.

6 Conclusion

In this thesis, we have seen what information was gathered for the development of the front-end and type checker. We looked at the syntax of MC, then we took a look at how parsers in general work which led to information on the parser monad. We saw how the parser for the MC mark 3 compiler works and found that the requirement for this project where met.

At the beginning of this project there was a research question: How to develop a maintainable and extendable front-end and type-checker for the MC mark 3 programming language?

During this project we found that the answer to that question is that the MC syntax allows for the front-end to be separated into several modules that allow for either parsing or type checking. Because the front-end is split into multiple modules the code of the front-end becomes more organized and easier to maintain. If functionality needs to be added to the front-end or type checker another module can be created and added to the system.

The code developed for the front-end, along with the documentation will further the goals of the research group to develop a language for game design.

7 Recommendations

7.1 Plans for the type function interpreter

During this project designs were made for the type function checker. These designs could however not be implemented during the time span of this project. The reason for this is that the type function checker would be the most complex part of the compiler. The type function checker needs to be able to interpret MC on all levels: terms, types and kinds. It would need to be part of the parsing processes or at least heavenly connected to it because type functions, or more precise modules, can be dependent on other type functions that have not been parsed yet. And the only way to parse those depended type functions is to first have its dependencies parsed and typed checked. We have here a circular dependency between the parser, type function checker and the compile time interpreter. What I would discourage is to make this system monolithic as it would go against the requirements that were set for the rest of the code. What I would suggest is to have the parts in this circular dependency turned into modules and have those modules interface with each other recursively. Those modules would be: A small parser that does not have to identify the function names of the definitions, a type function checker that can identify function names in the definitions and detect if those functions names call to a functions that has already been parsed and lastly an interpreter that works on kind level. The function checker module would be central in this system because it can decide to both: call the parser if a definition is not parsed yet and call the compile time interpreter to run functions. I think that a modular system for the type function checker would be needed if the research team wants a simple and maintainable type checker.

8 Evaluation

In this chapter we will look at the competence requirements associated with computer science according to the Rotterdam University of applied sciences.

8.1 Administering

8.2 Analyzing

During this project I often communicated with the stakeholders when new features were required in either the programming language or the compiler itself. When these new features needed to be added I prioritized them on importance and time required to implement. The importance of a feature I gathered from the stakeholders by looking at what they wanted that feature for.

8.3 Advising

I left documentation that described the plans I have for the future of this project so that the next research team can continue. Furthermore, in the code I left comments explaining the code and giving advise on how to expand on it.

8.4 Designing

I designed a front-end and type checker that would outlast my stay at this project. The type checker communicates with the back-end through a well defined interface that I helped develop and debug.