Seminar

Monadic Parser Combinators

Tutor: Prof.Dr.Michael Hanus

Author: Parissa H.Sadeghi

Date: May 2004

Christian-Albrechts-Universität zu Kiel

Contents

Abstract 3
1 Introduction 3
2 Functional Kernel 4
2.1 Pure Functional Programming 5
3 Monads and Computations 6
3.1 Monad 7
4 Functional Parsers 8
4.1 The Type of Parsers 9
4.1.1 Primitive Parsers 10
4.1.2 Parser Combinators 11
4.1.2.1 Sequence 11
4.1.2.2 Choice 14
4.1.3 Parsers and Monads 15
5 Lexical analyzer 16
6 Conclusions 16
6.1 Imitations 16
6.2 Advantages 18
7 A Simple Parser 19
8 References 23

Monadic Parser Combinators

Abstract

In functional programming, the parser can be written as a traditional recursive-descent parser using functions. Alternatively, one may use the monad style to set up the parser or attribute grammars.

In this paper, we will discuss about the meaning of monad and functional programming (is declared in [Kar98]) to provide functional parsers (is declared in [HM96]).

1 Introduction

Functional programming languages such as Haskell are eminent tools for defining *interpreters* and *compilers*. Program representations, in the form of *abstract syntax trees* (AST) are naturally defined used recursive data type declarations. Semantic operations are furthermore specified using features of a functional language such as patterns and equations and compositionality is in turn guaranteed in terms of lazy evaluation. The definitions of a *scanner* can be done very naturally by using the operations of the built in string type. The *parser* itself can be written as a traditional recursive-descent parser using Haskell functions, or alternatively, one may use the *monad style* to set up the parser or *attribute grammars*.

It was realised early on (Wadler, 1990) that parsers form an instance of a monad, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems. As well as being interesting from a mathematical point of view, recognising the monadic nature of parsers also brings practical benefits. For example, using a monadic sequencing *combinator* (or higher order functions) for parsers avoids the messy manipulation of nested tuples of results present in earlier work. Moreover, using *monad comprehension notation* makes parsers more compact and easier to read.

Construction of parsers for programming languages actually belong to one of the main stream applications of functional programming languages, since the application domain is itself purely functional: a parser is simply a function from text input to object code, which can be split up in intermediate compilation steps (functions) working on intermediate representations. The intermediate representation is conveniently represented in terms of a number of Haskell sum-of-product types, and the compilation steps can be elegantly expressed as functions, using features of the functional language such as equations, patterns, higher order functions etc.

In the area of tool integration, parser construction, or say lexical analysis, is frequently needed. Many CASE tools communicate with their environment by writing some text output to a file or to some interprocess communication medium. With a functional language at hand it becomes, compared to traditional technologies such as C combined with Lex and Yacc, very simple to write lexical analysers or parsers that peeks into the output generated by one tool, to extract information that will be needed by second tool.

2 Functional Kernel

Functional programming languages are recognized to have several nice properties. The notation offered is very close to that used in mathematics, and the semantic is well defined in terms of its foundation in lambda calculus [Bar84]. Moreover, functional languages are characterized by offering a powerful polymorphic type system, a useful module system, functions as first

class values, equational definition of functions using patterns and automatic garbage collection of values. Lazy functional languages add concepts such as evaluation of function parameters on need, which in turn form the basis for more speculative styles of problem solving based on e.g. infinite data structures. Haskell [HPW92] introduces classes and inheritance providing additional abstraction features almost similar to an objected style, with the limitation that binding is static and not dynamic as in an object oriented language.

2.1 Pure Functional Programming

Pure functional programming is a style of programming which consists entirely of the definition and application of functions [BW88, FH88, Wad92]. Unlike imperative languages there are no side-effects caused by the evaluation of expressions: the value of an expression is consequently independent of the context in which the expression occurs [Pey87].

The concept of *referential transparency* makes functional programs more applicable to rigorous, formal reasoning. Program transformations can more easily be performed on functional programs than an imperative one due to the absence of side-effects in functional programs. Furthermore, since functional programs are written in an equational style, it is possible to make the programs subject to proofs.

In addition, functional programming languages offer a range of powerful concepts at a high level of abstraction which makes them good vehicles for system development in a formal setting:

- A strong but flexible *type system* with *static typing*.
- *Type inference*, i.e. the ability of the system to infer the most general type of a value.
- Polymorphic functions and polymorphic data types.
- User defined algebraic datatypes.
- Powerful and predefined list type where lists can be defined using list *comprehensions*.
- *Higher order functions,* i.e. functions as first class values that take other functions as parameters or return functions as values.

- Equational definition of functions using pattern matching.
- User defined infix operators.
- Automatic memory allocation and de-allocation (*garbage collection*).
- And finally, a comprehensive set of predefined types and functions.

Some of these features are also offered, in one form or another, by modern imperative or object oriented languages. However, functional programming languages typically have more refined syntax than other languages and provide powerful features for abstraction which leads to programs being easier to write resembles mathematics more closely than imperative ones.

Functional programs therefore tend to be much shorter than programs written in imperative languages, although of course there is no guarantee that this will actually be the case.

Most functional languages are quite similar to each other. The major semantic distinction is the evaluation order of functions arguments which causes a classification of functional programming languages into two main categories.

Call by value: Eagerly evaluated languages evaluate the arguments to a function before the function is called. Standard ML (SML for short) [MTH90, Pau96, Bos95], Lisp [Ste90] and OPAL [DFG+94] are all eagerly evaluated languages.

Call by need: Lazily evaluated languages postpone the evaluation of an argument until its value is actually required in the body of a function. Clean [AP95], Miranda [Tur85], and Haskell [Hud2000, HPW92, PHA+, HPA+97, HPF97] are all lazily evaluated.

3 Monads and Computations

There have been several attempts [Gor92] over the last decades to extend a functional language with I/O and state, comprising approaches such as the *direct style* to I/O, *streams*, *continuation passing style*, *global state* and *monads*.

3.1 Monad

Imperative functional programming has its theoretical foundation in the *monadic style* of representing side effecting computations. Using *monads*, one may define the meaning of control flow operators of imperative programming languages, in particular those related to abnormal flow of control such as exceptions and goto's in a style that comes close to the continuation passing style used in specifying the formal semantics of imperative languages. Monads therefore bridge the gap between practical programming on one hand and formal specification technique on the other. The application area for monads does however go far beyond that of describing flow of control for sequential languages.

A **monad m a** is an abstract data type **m** of computations delivering a value of type **a**. The monad is represented in terms of triple (**m**, **return**, **>>=**) defined by a type constructor **m a** and a pair of polymorphic functions: **return** is the computation for returning a value of type **a** whereas (**>>=**) takes the role of *sequential composition*. The operator (**>>=**) combines a computation of type **m a** with a *continuation function* of type **a -> m b** to achieve a computation of type **m b**. In a composition **c>>=f**, **c** is executed first and the resulting value **v** is then passed to the continuation function **f**. The resulting computation is denoted by the expression (**f v**).

The type of two operators are defined as:

return :: a -> m a

(>>=) :: m a -> (a -> m b) -> m b

The monadic approach to I/O is based on structures from category theory [Wad90, Wad94, PW93], and can be used to represent *computations*.

As an example, in Haskell, the type **IO a** is a monad used for I/O. A computation of type **IO a** is a computation that will perform some side effecting I/O operations behind the scene. The visible effect of the computation is to return a value of type **a**, or eventually, fail with an error.

Now consider two functions: function **readNum** for reading a natural number from input and function **printNum** for displaying the value of each expression entered into the interpreter :

```
readNum  :: IO Int
printNum  :: Int -> IO ()
```

And suppose we want to write a main program for reading a number from input and computing its factorial that will be written to standard output, as follows:

```
main ::  IO ()
main =  readNum  >>=  \v  ->  printNum ( fac v )
```

We can write this main program with *do notation*. The do notation is actually shorthand for use of the monadic operators. Using do notation it will be possible to write imperative functional programs involving assignment statements, sequencing of actions and compound actions in a style that is very close to the way it is usually done in traditional imperative languages:

```
main   ::  IO ()

main  =  do

      v  <-  readLn

      print  (fac v)
```

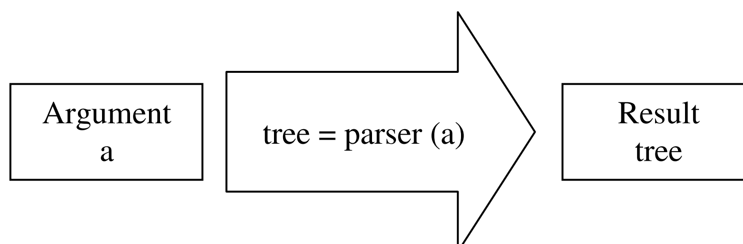
The symbol (**<-**) serves as a single assignment operator to accept patterns on the right hand side.

4 Functional Parsers

G.Hutton and E.Meijer have provided a step-by-step tutorial on the monadic approach to building functional parsers and to explain some of the benefits that result from exploiting monads. Therefore we will use their papers to define a type for parsers, three primitive parsers, and two primitive combinators for building larger parsers.

4.1 The Type of Parsers

At the first, we need to define a parser as a function, which takes a string of characters as input and yields some kind of syntax tree as result:



Since, a parser might not consume all of its input string, we must return the unconsumed suffix of the input string too. We know also that a parser might fail on its input and some times the parsers are ambiguous i.e. they can produce more than one tree for one input string. Thus, the result of a parser must be a *list* of consumed and unconsumed of the input string:

type Parser a = String -> [(a , String)]

As an example, we consider a parser that is defined with the grammatical structure to recognize a list of **x** and is an ambiguous grammar:

P -> P x P | ε   =>   L(P) = { ε, x, xx, … }

This parser can be defined as a functional parser with the type of our defined parser. If for example, we give difference strings to this parser then a list of consumed and unconsumed of input string will be as follows:

input : "xx"   =>   output : [(Tree1 "xx" , ""), (Tree2 "xx" , "")]

where:

Tree1"xx" :   P -> PxP -> xP -> xPxP -> xxP -> xx

Tree2"xx" :   P -> PxP -> PxPxP -> xPxP -> xxP -> xx

input : "xy"   =>   output : [(Tree1"x","y")]

where:

Tree1"x" : P -> PxP -> xP -> x

This type of parser is used in paper of Hutton and Meijer to define a functional parser.

4.1.1 Primitive Parsers

Now we know how the type of our parser is. But we must declare some functions to have different parser combinators. The first primitive parser is **result v** that takes an input string **inp** and returns the list **[(v,inp)]**. It succeeds without consuming any of the input string, and returns the single result **v** :

```
result   ::  a -> Parser a
result v = \ inp -> [ ( v , inp ) ]
```

We can compare this parser with the grammatical structure:

P -> ε

The second primitive parser is **zero** that always fails, regardless of the input string :

```
zero :: Parser a
zero = \ inp -> []
```

That means, for every input string, the result of **zero** is an empty list.

The final primitive parser is **item**, which successfully consumes the first character if the input string is non-empty, and fails otherwise:

```
item :: Parser Char
item = \ inp -> case inp of
                    []      ->  []
                    (x:xs) ->  [(x,xs)]
```

As an example, if the string *"aab"* is an input for the defined parser, then parser **item** will return *[('a',"ab")]* as result and parser **result** will have another list [("","aab")] , but parser **zero** fails.

4.1.2 Parser Combinators

The three primitive parsers are not very useful in themselves. Thus Hutton and Meijer decided to use some operators, like to *sequence* and *choice*, to combine parsers.

4.1.2.1 Sequence

In *non-monadic* accounts of combinators, sequencing of parsers was usually captured by a combinator, when it applies one parser after another parser, with the results from the two parsers being combined as pairs:

```
seq         :: Parser a -> Parser b -> Parser (a,b)
p `seq` q  = \ inp -> [ ((v,w),inp")  |   (v,inp')  <- p inp
                                       ,   (w,inp") <- q inp' ]
```

The right hand side of sign **|** means: if parser **p** takes input string **inp**, then the consumed input as a tree together with the unconsumed input string will be return in a list of pair. Each pair is declared as **(v,inp')**. After that, parser **q** takes **inp'** as input and returns a list of pair with the consumed part of **inp'** as a tree and unconsumed part of **inp'**. One element of this list is **(w,inp")**. Finally, combinator **seq** returns a combination between two results value **v** , **w** that is a list of [((v,w) , inp")].

But, in practice, using **seq** leads to parsers with nested tuples as results, which are messy to manipulate.

The problem of nested tuples can be avoided by adopting a *monadic* sequencing combinator **bind**, which integrates the sequencing of parsers with the processing of their result value:

```
bind        :: Parser a -> (a -> Parser b) -> Parser b
p `bind` f = \inp -> concat [ f v inp'  |  (v,inp') <- p inp ]
```

Combinator **bind** defined above has two inputs, a parser and a parser abstraction. First of all, the parser **p** applies to the input string and gives a result that is a list of (value, string) pairs [section 4.1]. Now function **f** applies to all of the values and returns a parser. The new parser is a list of concatenated lists.

Since the result of a parser is available for processing by the next parser, the **bind** combinator avoids the problem of nested tuples of results.

For example, the ***seq*** combinator can be defined by:

```
seq :: Parser a -> Parser b -> Parser (a,b)

p  `seq`  q  =  p  `bind`  \x  ->

                q  `bind`  \y  ->
                result (x,y)
```

That means: apply parser **p** and denote its result value **x**, then apply parser **q** and denote its result value **y**, and finally combine all the result into a single value by applying function **seq**. (But,in practice **bind** can not be defined in terms of **seq**.)

And in grammatical structure, if for example we declare a list of **a** that followed with a list of **b** :

S -> P Q

P -> a P | ε

Q -> b Q | ε

The result of parser *S* will be a concatenation of two lists. Thus for an input string like to *"aab"*, the parser will return a concatenation between strings of two lists that is *[(("a",""),"ab"),(("aa","b"),""), (("aa",""),"b"), (("",""),"aab")]*.

With **bind** combinator, we can define some different and useful parsers. As an example we define a combinator **sat** that takes a Boolean valued function as a predicate and yields a parser that consumes a single character if it satisfies

the predicate and fails otherwise. For this approach we use our simple parsers: parser **item**, that consumes a single character unconditionally and parser **zero** that is defined for returning an empty list, when parser fails:

```
sat    ::  (Char -> Bool)  -> Parser Char
sat p =  item  `bind`  \x  ->  if p x  then  result x
                                          else  zero
```

In this declarated combinator, if the input string is empty or first letter of input is not a character, then parser **item** fails by **zero**, otherwise, it will return a list of value and unconsumed string by **result**.

Using **sat**, we can define parsers for specific characters, single digits, lower-case letters and upper-case letters:

```
char     ::  Char  ->  Parser  Char
char x  =  sat ( \y  ->  x == y )

digit  ::  Parser  Char
digit  =  sat ( \x  -> ‘0’ <=  &&  x <= ‘9’ )


lower  ::  Parser  Char
lower  =  sat ( \x  -> ‘a’ <= x  &&  x <= ‘z’ )


upper  ::  Parser Char
upper  =  sat ( \x -> ‘A’ <= x  &&  x<= ‘Z’ )
```

For example, we can use **bind** combinator to return a string with two characters:

```
lower  `bind`  \x  ->
lower  `bind`  \y  ->
result [ x,y ]
```

Applying this parser to the string *"abcd"* succeeds with the result *[("ab" , "cd")]* but it fails with the result *[ ]* when *"aBcd"* is the input, because the second letter *'B'* can not be consumed by the second **lower** parser.

Since the length of the string to be parsed can not be predicted, in general we will need a recursive parser. We know every recursive function must be terminated, thus a **choice** operator will help us to decide between parsing a single letter and recursing, or parsing nothing further and terminating.

4.1.2.2 Choice

A suitable **choice** combinator for parsers is defined as **plus**:

```
plus         ::  Parser a  -> Parser a  -> Parser a
p `plus` q  =  \inp  ->  ( p  inp  ++  q  inp )
```

Combinator **plus** takes two parsers as input that both of them apply to the same input string. Each parser **p** or **q** has a list as result that will be concatenated to form a single result list. But there is a problem, if the first parser accepts the input string then the second parser will be applied to the same input, even though it is guaranteed to fail. That means, we need an extra run-time for processing the second parser. While, if the first parser succeeds on the input string, we have the result and combinator must be terminated. Because of that, we define a new *choice* **(+++**) which terminates with the first success:

```
(+++)      ::  Parser a  -> Parser a  -> Parser a

p +++ q  =  first (p `plus` q)

first        ::  Parser a  -> Parser a

first p      = \inp   -> case  p inp  of

                              []       ->  []

                              (x:xs) ->  [x]
```

The parser **first p** returns only the first result (if any). That means, using **first** we have defined a *deterministic* version (+++) of the standard choice combinator for parsers.

As an example, if parser *P* recognizes a list of character *'a'* and parser *Q* recognizes a list of character *'b'*, we can combine parser *P* and *Q* to define a new parser that returns a list of *'a'* or a list of *'b'* :

```
S  ->  P  |  Q        =>    s = p `plus` q

P  ->  a P  |  a      =>    p = ((char 'a')  `seq`  p)  `plus` ( char 'a')

Q  ->  b Q  |  b      =>    q = ((char 'b' )  'seq'  q) `plus` ( char 'b')
```

Now we can combine our earlier defined parsers and combinators to define larger parsers. As an example, a parser that accepts letter and recognizes a word (strings of letters):

```
letter  ::  Parser  Char
letter  =  lower  `plus`  upper

word  ::  Parser  String
word  =  neWord  `plus`  result ""
           where
                neWord = letter  `bind`  \x   ->
                           word  `bind`  \xs ->
                           result ( x : xs )
```

4.1.3 Parsers and Monads

We have seen in section 3.1that a *monad* is a type constructor with two operations **return** and **>>=** . And we have said in section 4.1 that functional parsers have a **Parser** type constructor. Thus, parsers form a monad.

Hutton and Meijer have compared a *parser* with the *monad* type constructor. Thus, the following operations can be introduced for the parser type constructor (they have used operation **result** as operation **return** and operation **bind** as operation **>>=**):

result :: a -> Parser a

bind :: Parser a -> (a -> Parser b) -> Parser b

With this definition, we can say, our defined parser is a monadic parser.

5 Lexical analyzer

Traditionally, first every input string will be passed through a lexical analyser (scanner) that breaks the string into a sequence of tokens. And after that, it will be parsed by a syntax analyser (parser).

Spaces, newlines, tabs and comments will be removed by scanner and identifiers and also keywords will be distinguished. But using parser combinators, we don't need a separate scanner. Only with writing some combinators, we can add a scanner within the main parser.

6 Conclusions

Now, with the primitive parsers and combining the primitive combinators, we can write a lager parser to recognize every input string. But, what is the advantages and imitations in the monadic parser combinators:

6.1 Imitations

An important restriction on most existing combinator parser is that they are unable to deal with *left-recursion* [LM2001]. But in practice, grammars are often left-recursive. Fortunately, every left-recursive grammar can be written into a right-recursive one.

In practice, a parser combinator refers a list of successes as result, since the parser combinator protects ambiguous grammars. It doesn't matter whether

that the list contains zero, one or many elements. They are all valid answers. But it makes it hard to give a good error message.

That means, another problem is *error-reporting*. A good parser's error message contains the position of the error in the input and also the cause of the error. Beside error reporting, the parser might try to correct the error. But unfortunately, current parser combinators (nondeterministic) are not very good at reporting errors. However, even when we restrict ourselves to non-ambiguous grammars, it is hard to report an error since the parsers can always look arbitrary far ahead in the input (they are LL(∞)) and it becomes hard to decide what the error message should be.

Because of that, Leijen and Meijer in *Parsec* [LM2001] restricted themselves to *deterministic* predictive parsers with limited lookahead, for example LL(1) parser (but, most grammars are not LL(1) and even require arbitrary lookahead).

They consider that a parser has either **Consumed** input or returns a value without consuming input (**Empty**). The return value is either a single result and the remaining input (**Ok a String**), or a parse error (**Error**). Thus, they have defined a new parser type constructor with error message, as follows:

```
type Parser  a  =  String  -> Consumes a

data Consumed  a  =  Consumed (Reply  a)  |   Empty (Reply a)

data Reply  a  =  Ok a String  |  Error
```

For example, the first primitive parser **result** that has been defined in section 4.1.1 can be written with this method:

```
result  v  =  \inp -> Empty (Ok  v  inp)
```

The **result** succeeds immediately without consuming any input string, hence it returns the Empty alternative.

6.2 Advantages

So far we have seen one advantage of recognising the monadic nature of parsers: the monadic sequencing combinator **bind** handles better than the traditional sequencing combinator **seq**.

Another advantage of the monadic approach, namely that *monad comprehension syntax* can be used to make parsers more compact and easier to read. For example, earlier we have defined a **sat** combinator as follows:

```
sat      :: (Char -> Bool) -> Parser Char
sat   p = item `bind` \x ->
               If p x then result x else zero
```

Gofer [Jon94] provides a special notation for defining parsers of this shape, allowing them to be expressed in the following, more appealing form:

```
sat     :: (Char -> Bool) -> Parser Char
sat  p =  [x |  x <- item , p x]
```

In fact, this notation is not specific to parsers, but can be used with any monad [Jon95]. There is another notation that can be used to make monadic programs easier to read. This notation is defined in [JSL94] and is named as **do** notation:

```
sat      :: (Char -> Bool) -> Parser Char
sat  p  =  do { x <- item ; if (p x) ; result x }
```

7 A Simple Parser

To illustrate the monadic parser combinators, we write a simple parser to define a language as follows:

$L(S) = \{ a^n b^n c^n \mid n \in Z \}$

That means, this language can accept only a list of three characters a, b and c, when the number of these characters are equal and when they have a sequential relationship between themselves. A simple context-free grammatical structure of this language can be written as follows:

S -> A B C
A -> a A | ε
B -> b B | ε
C -> c C | ε

To write a monadic parser, we must use our defined type parser and primitive parsers, which are defined in section 4.

Since this language needs to recognize three charactes a, b and c , combinator **sat** and **char** will help us to complete one part of program.

```
sat          :: (Char -> Bool)  ->  Parser Char
sat   p    =  item  `bind`   \x ->
                        if  p x  then  result x  else  zero
char        :: Char       ->  Parser Char
char   x      =  sat ( \y   ->  x == y )
```

To define the number of characters in the input string we need a recursive function. Function **count** takes a parser, applies it **n** times and returns the parser **result** as output :

```
count         :: Parser a -> Parser Int
count   p     = ( p         `bind`    \_   ->
                      count  p `bind`   \n   ->
                                       result (n+1) )  `plus`  (result 0)
```

If the input string of parser **p** has no character, then the output of **count** will be **result 0**, but if it has **n** character, the output will be parser **result n**.

Now we must define our grammatical structure as a function. Function **s** is written for grammatical structure of nonterminal **S**. The output of this function is True, when the number of three character equal is i.e. na=nb=nc :

```
s     :: Parser Bool
s     =  count (char 'a')  `bind`  \na  ->
         count (char 'b')  `bind`  \nb  ->
         count (char 'c')  `bind`  \nc  ->
                                   result ((na==nb) && (na==nc))
```

We can define a function to check the input string with the defined language (function **startParser**):

```
startParser           :: Parser a  ->  String  ->  Maybe a
startParser  p  str   =  find (p  str)
        where   find  []              = Nothing
                      find  ((r , "") : _ ) = Just r
                      find   (( _ : res))  = find  res
```

Function **find** will take the result of parser **p** (that is a list) to check the situation of the list. If the result of **p** is an empty list, this means that the input string is not a member of the language (**Nothing**). But if the list is non-empty, the input string can be produced by grammar, but either the number of characters a, b and c are equal (**Just True**) or are not equal (**Just False**).

The results of simple defined parser for difference inputs are :

Input : “” => output : Just True
Input : “ab” => output : Just False
Input : “aabbcc” => output : Just True
Input : “abbcca” => output : Nothing

To show the difference between monadic and non-monadic combinators sequence that is defined in section 4.1.2.1. We change the function **count** to produce parser **result** that returns a list of characters (**count2)**:

```
count2      ::  Parser Char  ->  Parser String
count2  p  = (p                `bind`   \c  ->
                 count2   p   `bind`   \cs  ->
                                      result (c:cs))  `plus`  (result"")
```

Now we try to write two difference function for nonterminal **S** with monadic sequence **bind** and non-monadic sequence **seq** (section 4.1.2.1):

```
s1    ::   Parser String
s1   =   ( count2 (char  'a'))  `bind`   \na ->
         ( count2 (char  'b'))  `bind`   \nb ->
        ( count2 (char  'c') ) `bind`   \nc  ->
                                      result(na++nb++nc)
```

```
s2    ::  Parser ((String,String ), String)
s2   =  (count2 (char  'a'))  `seq`
        (count2 (char  'b'))  `seq`
       (count2 (char  'c'))
```

The results of the two parser functions **s1** and **s2** with different input string will show us the problem of nested-tuples:

Input : “”   => output (s1) : [("","")]

Input : “”   => output (s2) : [((("",""),""),"")]

Input : “ab” => output (s1) : [("ab",""),("a","b"),("","ab")]

Input : “ab” => output (s2) :
[((("a","b"),""),""),((("a",""),""),"b"),((("",""),""),"ab")]

Input : “aabbcc” => output (s1) :
[("aabbcc",""),("aabbc","c"),("aabb","cc"),("aab","bcc"),("aa","bbcc"),("a"," abbcc"),("","aabbcc")]

Input : “aabbcc” => output (s2) :

[((("aa","bb"),"cc"),""),((("aa","bb"),"c"),"c"),((("aa","bb"),""),"cc"),((("aa","b"),""),"bcc"),((("aa",""),""),"bbcc"),((("a",""),""),"abbcc"),((("",""),""),"aabbcc")]

Input : “abbcca” => output (s1) :

[("abbcc","a"),("abbc","ca"),("abb","cca"),("ab","bcca"),("a","bbcca"),("","abbcca")]

Input : “abbcca” => output (s2) :

[((("a","bb"),"cc"),"a"),((("a","bb"),"c"),"ca"),((("a","bb"),""),"cca"),((("a","b"),""),"bcca"),((("a",""),""),"bbcca"),((("",""),""),"abbcca")]

This shows that the result of non-monadic style parser is more difficult to process due to the nested tuple structure.

8 References

[AP95] P. Achten and R. Plasmeier. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 1995.

[Bar84] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North Halland, 1984.

[Bos93] R. Bosworth. *APractical Course in Functional Programming Using ML*. 1995.

[BW88] B. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[DFG+94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of Lecture Notes Computer Science. Springer Verlag, March 1994.

[FH88] A. J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[Gor92] A.D.Gordon. *Functional Programming and Input/Output*. PhD thesis, Computer Laboratory, University of Combrige, England, 1992.

[HM96] G. Hutton and E.Meijer. *Monadic Parser Combinators*. Appears as technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[HPA+97] K. Hammond, J. Peterson, L. Augustsson, J. Fasel, A. D. Gordon, S. Peyton Jones, and A. Reid. *Standard Libraries for the Haskell Programming Language, Version 1.4*, April 1997.

[HPF97] P. Hudak, J. Peterson, and J. H. Fasel. *A Gentle Introduction to Haskell* – Version 1.4 - , March 1997.

[HPW92] P.Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell – a non strict purely functional language, version 1.2. ACM SIGPLAN *notices*, 1992.

[Hud2000] P. Hudak. The Haskell School of Expression, Learning Functional Programming Through Multimedia. 2000.

[Jon94] Jones and P. Mark. *Gofer 2.30a release notes*. Unpublished manuscript.1994.

[Jon95] Jones and P. Mark. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming, 1995.*

[JSL94] Jones, P. Simon, and J. Launchbury. *State in Haskell*. University of Glasgow. 1994.

[Kar98] Einar W. Karlsen. *Tool Integration In A Functional Programming Language*. M. Gogolla, H. -J. Kreowski, B. Krieg-Brückner, J. Peleska, B. –H. Schlingloff, H. Szczerbicka (Series Editors), 1998.

[LM2001] D. Leijen and E. Meijer. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. October, 2001.

[MTH90] R. Milner, M. Tofte, and R.Harper*. The Definition of Standard ML*. MIT Press, 1990.

[Pau96] L. C. Paulson.*ML for the Working Programmer*. Cambrige University Press, 2. edition, 1996.

[PHA+97] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming Languages Haskell*, Version 1.4, January 1, 1997.

[PW93] S. Peyton Jones and P. Wadler. Imperative Funczional Programming. In *Proc. 20th ACM Symposium on Principles of Functional Programming*, 1993.

[Ste90] G. L. (Jr) Steele. *Common Lisp the Language*. Digital Press, 2 edition, 1990.

[Tur85] D. A. Turner. Miranda: A non-strict Functional Language with Polymorphic Types. In *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

[Wad90] P. Wadler. Comprehending Monads. In ACM *Conference on Lisp and Functional Programming*, 1990.

[Wad92] P. Wadler. The essence of Functional Programming. In ACM *Principles of Programming Languages*, 1992.

[Wad94] P. Wadler. Monads and Composable Continuations. *Lisp and Symbolic Computation*. January 1994.