EDAF95/EDAN40: Functional Programming
Monadic Parsing

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

May 8th, 2023

What is a parser?

Hutton:

A *parser* is a program that takes a string of characters, and produces some form of tree that makes the *syntactic structure* of the string explicit.

Example: 2 * 3 + 4

Note ambiguity!

An example of a program


```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')){}#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

An example of a program

“Twelve Days of Christmas” song

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')){}#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Some other language

```
-----------------------------------------
 module Main where{import List;import System;
 import Data.HashTable as H;(???????)=(concat
 );(??????)(???)(????)=((groupBy)(???)(????))
 ;(????????????????????)(????)=((??????????
 )((tail).(???????))((????????????????????)((
 ??????)(????????????????????)(????))));(??)
 =([' ']);(????????????????)=((hashString));(?)
 =((>>=));(????????????????????)([((???)),
 (????)])=((????????????????)(???))?(\(?????)->(
 (????????????????)(==)(????????????????))?(\((
 ???))->((????????????????????)(????????????????
 )(???)(?????))>>((????????????????????)(???))?(
 \((?????))->((((????????????????????)((????))
 ((????????????????????????))((?????))))))));((
 ????????????????????????))(??)=(???????????)
 ("usage f dic out");(????????????????????)(
 (???),(??????))((????),(????????????????????
 ))=((???)==(????));(????????????????)(???)=
```

Some other language

```
(toList)(???);(??????????????????)(????)=(
((??????????)(((??????????)(snd)))((????))))
;(???????????????)(????????????????)(???)(
(?????))=(((mapM)(((????????????????)(???)))(
(lines)(?????))));(???????????????????)(????
)(????????????????????????)(?????)=(?????????
)(????)((unlines)((??????????????????????)(
?????)));(???????????????)(???)((????))=(((
new)(???)(????)));(main)=((??????????)?(((\
(???)->((??????????????????????)(???))))));
(????????????????)(???)(????)=((????????)(???
)((sort)(????))((??)++(????)));(??????????)
=(getArgs);(???????????)(???)=((((print))((
???))));(??????????)(???)(????)=(((map)(???)
(????)));(????????)((???))(????)(?????)=((((
H.insert))((???))(????)(?????)));(?????????)
((???))((????))=(((writeFile)(???)((????))))
;(????????????)(???)=(((readFile)((???))))}
----------------------------------------
```

What is parsing?

Analysis of a text in order to:

- decide whether it is correct,
- decide its internal structure,
- decide its meaning,
- manipulate it somehow.

Normally the first two steps are considered *parsing*. In the context of assignment N3 the other two will be treated as *interpretation*.

Correctness (of a string)

- Formal languages
- Grammars
- Automata

Chomsky hierarchy

The language

```
read k;
read n;
m := 1;
while n-m do
  begin
    if m - m/k*k then
      skip;
    else
      write m;
    m := m + 1;
  end
```

And its grammar

```
program ::= statements
statement ::= variable ':=' expr ';'
  | 'skip' ';'
  | 'begin' statements 'end'
  | 'if' expr 'then' statement
              'else' statement
  | 'while' expr 'do' statement
  | 'read' variable ';'
  | 'write' expr ';'
statements ::= {statement}
variable ::= letter {letter}
```

The parser type

Assume some suitable definition of a `Tree` type. Then:

```
type Parser = String -> Tree
```

The parser type

Assume some suitable definition of a `Tree` type. Then:

```
type Parser = String -> Tree
```

In general, a parser will not consume a complete string. So:

```
type Parser = String -> (Tree, String)
```

The parser type

Assume some suitable definition of a `Tree` type. Then:

```
type Parser = String -> Tree
```

In general, a parser will not consume a complete string. So:

```
type Parser = String -> (Tree, String)
```

Let's denote a failure to parse by the empty list, and a success by a singleton:

```
type Parser = String -> [(Tree, String)]
```

The parser type

Assume some suitable definition of a `Tree` type. Then:

```
type Parser = String -> Tree
```

In general, a parser will not consume a complete string. So:

```
type Parser = String -> (Tree, String)
```

Let's denote a failure to parse by the empty list, and a success by a singleton:

```
type Parser = String -> [(Tree, String)]
```

Finally, let us allow any kind of output values, e.g. numbers:

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

Basic parsers

Always succeeds with the result value `v`:

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

Always fails:

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

Fails on empty input; succeeds with the first character otherwise:

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

Parser application

Parser application function:

```
parse :: Parser a -> String -> [(a, String)]
parse p inp = p inp
```

For example:

```
> parse (return 1) "abc"
[(1, "abc")]
> parse failure "abc"
[]
> parse item ""
[]
> parse item "abc"
[('a',"bc")]
```

Sequencing

The sequencing (bind) operator:

```
(>>=)    :: Parser a -> (a -> Parser b) -> Parser b
p >>= f  = \inp -> case parse p inp of
                     [] -> []
                     [(v, out)] -> parse (f v) out
```

A typical parser built this way will have the following structure:

```
p1 >>= \v1 ->
p2 >>= \v2 ->
...
pn >>= \vn ->
return (f v1 v2 ... vn)
```

Sequencing, cont.

Syntactic sugar for this is:

```
do v1 <- p1
   v2 <- p2
   ...
   vn <- pn
   return (f v1 v2 ... vn)
```

Expressions `vi <- pi` are called *generators*. If the result is not required, the generator may be abbreviated as `pi`.

Sequencing, cont.

Example: a parser that returns the first and third character of a string as a pair and discards the second character:

```
p :: Parser (Char, Char)
p = do x <- item
       item
       y <- item
       return (x,y)

> parse p "abcdef"
[(('a', 'c'), "def")]

> parse p "ab"
[]
```

Choice

orElse:

```
infixr 5 +++

(+++) :: Parser a -> Parser a -> Parser a
p +++ q = \inp -> case parse p inp of
                    [] -> parse q inp
                    [(v,out)] -> [(v,out)]
```

Derived primitives

First, a parser satisfying a predicate:

```
sat     :: (Char -> Bool) -> Parser Char
sat p   =  do x <- item
              if p x then return x else failure
```

Derived primitives, cont.

```
digit          :: Parser Char
digit          =  sat isDigit

lower          :: Parser Char
lower          =  sat isLower

upper          :: Parser Char
upper          =  sat isUpper

letter         :: Parser Char
letter         =  sat isAlpha

alphanum       :: Parser Char
alphanum       =  sat isAlphaNum
```

Derived primitives, cont.

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

string         :: String -> Parser String
string []      =  return []
string (x:xs)  =  do char x
                     string xs
                     return (x:xs)
```

Derived primitives, cont.

```
many           :: Parser a -> Parser [a]
many p         =  many1 p +++ return []

many1          :: Parser a -> Parser [a]
many1 p        =  do v  <- p
                     vs <- many p
                     return (v:vs)

ident          :: Parser String
ident          =  do x  <- lower
                     xs <- many alphanum
                     return (x:xs)
```

Derived primitives, cont.

```
nat             :: Parser Int
nat             =  do xs <- many1 digit
                      return (read xs)

int             :: Parser Int
int             =  do char '-'
                      n <- nat
                      return (-n)
                   +++ nat

space           :: Parser ()
space           =  do many (sat isSpace)
                      return ()
```

Handling spacing

```
token          :: Parser a -> Parser a
token p        =  do space
                     v <- p
                     space
                     return v

identifier     :: Parser String
identifier     =  token ident
```

Handling spacing

```
natural        :: Parser Int
natural        =  token nat

integer        :: Parser Int
integer        =  token int

symbol         :: String -> Parser String
symbol xs      =  token (string xs)
```

Arithmetic expressions: grammar

```
expr   ::= term expr'
expr'  ::= addOp term expr' | empty
term   ::= factor term'
term'  ::= mulOp factor term' | empty
factor ::= num | var | "(" expr ")"
addOp  ::= "+" | "-"
mulOp  ::= "*" | "/"
```

Parsing expressions

```
expr   :: Parser Int
expr   =  do t <- term
             do symbol "+"
                e <- expr
                return (t+e)
              +++ return t

term   :: Parser Int
term   =  do f <- factor
             do symbol "*"
                t <- term
                return (f * t)
              +++ return f
```

Parsing expressions

```
factor :: Parser Int
factor =  do symbol "("
             e <- expr
             symbol ")"
             return e
          +++ natural

eval     :: String -> Int
eval xs =  case (parse expr xs) of
             [(n,[])]  -> n
             [(_,out)] -> error ("unused input " ++ out)
             []        -> error "invalid input"
```

The conclusion

```
newtype Parser a =  P (String -> [(a,String)])

instance Monad Parser where
   return v    =  P (\inp -> [(v,inp)])
   p >>= f     =  P (\inp -> case parse p inp of
                               []        -> []
                               [(v,out)] -> parse (f v) out)
```

The conclusion

```
newtype Parser a =  P (String -> [(a,String)])

instance Monad Parser where
   return v    =  P (\inp -> [(v,inp)])
   p >>= f     =  P (\inp -> case parse p inp of
                               []        -> []
                               [(v,out)] -> parse (f v) out)

instance MonadPlus Parser where
   mzero       =  P (\inp -> [])
   p `mplus` q =  P (\inp -> case parse p inp of
                               []        -> parse q inp
                               [(v,out)] -> [(v,out)])
```

Then we have:

```
failure          :: Parser a
failure          =  mzero

parse            :: Parser a -> String -> [(a,String)]
parse (P p) inp  =  p inp

(+++)            :: Parser a -> Parser a -> Parser a
p +++ q          =  p `mplus` q
```

Real World Haskell

The parsers in Chapter 10 use two variants:

- the `Maybe` monad;
- the `Either` monad.

Chapter 16 introduces `Parsec`: a parser combinator library. Suitable both for

- lexical analysis (flex domain)
- actual parsing (bison domain)

See also lecture notes on parsing by Lennart Andersson (linked to from the Assignment 3 page).

Packrat parsing

- Grammar:
  PEG (Parsing Expression Grammar) instead of CFG (context-free grammar)
  introduced in 2004, Bryan Ford

Packrat parsing

- Grammar:
  PEG (Parsing Expression Grammar) instead of CFG (context-free grammar)
  introduced in 2004, Bryan Ford
- Behaviour:
  guaranteed *linear time*!
  due to elimination of ambiguity from the grammar

Example PEG for expressions

```
Expr    <- Sum
Sum     <- Product (('+' / '-') Product)*
Product <- Value (('*' / '/') Value)*
Value   <- [0-9]+ / '(' Expr ')'
```

Assignment N3

- Complete a (`Maybe`-based) parser in order to parse a program in a simple language,
- Write an interpreter for this language.

The language

```
read k;
read n;
m := 1;
while n-m do
  begin
    if m - m/k*k then
      skip;
    -- this is my comment
    else
      write m^2;
    m := m + 1; -- a comment as well
  end
```

And its grammar

```
program ::= statements
statement ::= variable ':=' expr ';'
  | 'skip' ';'
  | 'begin' statements 'end'
  | 'if' expr 'then' statement
              'else' statement
  | 'while' expr 'do' statement
  | 'read' variable ';'
  | 'write' expr ';'
statements ::= {statement}
variable ::= letter {letter}
```

Assignment F2

Parse:

```
030050040
008010500
460000012
070502080
000603000
040109030
250000098
001020600
080060020
========
```