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!
Monadic Parsing

An example of a program

"Twelve Days of Christmas" song

```c
#include <stdio.h>
main(_,_,a)char *a;{return0<t?c3main(-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=27<_13?
main(2,_,1,%XdXdln):9:16:<0?t<77?main(_,t,
"gn'+'*/jv/w&chrn',j/r/oe)/+/s(+j/*,wqKm+a,+#1,+,n[n,+/#n,+/n|
&gqua:,+/kx=,+/r:='d3,(uK w'k'w'+1)#k'dq'1 l \
q8d'k#r'KeX(#r)eKH(nl)#h#qK'#/h#y)(ml)/"#n;1:(atv i# \
)(!1!)(n(n#: r@v'r nc(nl)#1,+#v'k iK{[n(n'/wqKm+wX nx
wKX(ml)/w"1!#wv' i :;
/ml)/(*q8'ld;r')nlvb1/*de)c \
;nl1!"-]w'/"h#s*mc",,'nu1"/kd'!'e)+;#rdqKv! nr'/ ') }+{1(1'n '1" \ 
') }+{1(1"
):c<-507==a?putchar(31[a]:main(-65,_,a+1)main((a='/')+(t,_,a+1) 
:0?main(2,2,"%a":a='/'|main(0,main(-61,*,
"f[k;dc i0hK'q-){x[2]*x[3]l_i:}nuxloca-0;n.vpbks,fxtndCeghiy'),a+1);
```

Jacek Malec, http://rss.cs.lth.se

---

Some other language

module Main where(import List;import System; 
import Data.HashTable as H;(???????)=(concat 
(???????)???)=(??))

```
(???????)=(??)
```

Jacek Malec, http://rss.cs.lth.se

---

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.
Monadic Parsing

Correctness (of a string)

- Formal languages
- Automata
- Grammars

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
```

Jacek Malec, http://rss.cs.lth.se
The parser type

Assume some suitable definition of a Tree type. Then:

\[
\text{type Parser } = \text{String } \rightarrow \text{Tree}
\]

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

\[
\text{type Parser } = \text{String } \rightarrow (\text{Tree, String})
\]

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

\[
\text{type Parser } = \text{String } \rightarrow [(\text{Tree, String})]
\]

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

\[
\text{type Parser a } = \text{String } \rightarrow [(a, \text{String})]
\]

Basic parsers

Always succeeds with the result value \(v\):

\[
\text{return :: a } \rightarrow \text{Parser a}
\]

\[
\text{return v } = \inp \rightarrow [(v, \inp)]
\]

Always fails:

\[
\text{failure :: Parser a}
\]

\[
\text{failure } = \inp \rightarrow []
\]

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

\[
\text{item :: Parser Char}
\]

\[
\text{item } = \inp \rightarrow \text{case inp of}
\]

\[
[] \rightarrow []
\]

\[
(x:xs) \rightarrow [(x, xs)]
\]
Parser application

Parser application function:

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

For example:

\[\begin{align*}
& \text{parse (return 1)} "abc" \\
& \text{parse failure "abc"} \\
& \text{parse item ""} \\
& \text{parse item "abc"} \\
\end{align*}\]

Jacek Malec, http://rss.cs.lth.se

Sequencing

The sequencing (bind) operator:

\[\begin{align*}
(\gg&=) \quad : \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b \\
p &\gg= f = \text{ \inp } \rightarrow \text{ case parse } p \text{ \inp } \text{ of} \\
&\quad [] \rightarrow [] \\
&\quad [(v, \text{ out})] \rightarrow \text{ parse } (f v) \text{ out} \\
\end{align*}\]

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

\[\begin{align*}
p1 &\gg= \text{ \v1 } \rightarrow \\
p2 &\gg= \text{ \v2 } \rightarrow \\
&\quad \ldots \\
pn &\gg= \text{ \vn } \rightarrow \\
\text{ return } (f \text{ v1 v2 } \ldots \text{ vn}) \\
\end{align*}\]

Sequencing, cont.

Syntactic sugar for this is:

\[\begin{align*}
do \ v1 &\leftarrow p1 \\
&\quad v2 &\leftarrow p2 \\
&\quad \ldots \\
&\quad \vn &\leftarrow pn \\
\text{ return } (f \text{ v1 v2 } \ldots \text{ vn}) \\
\end{align*}\]

Expressions \(vi \leftarrow pi\) are called *generators*. If the result is not required, the generator may be abbreviated as \(pi\).

Jacek Malec, http://rss.cs.lth.se
orElse:

infixr 5 +++

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

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

digit = sat isDigit

lower :: Parser Char
lower = sat isLower

lower :: Parser Char
lower = sat isLower

upper :: Parser Char
upper = sat isUpper

upper :: Parser Char
upper = sat isUpper

letter :: Parser Char
letter = sat isAlpha

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

alphanum :: Parser Char
alphanum = sat isAlphaNum

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)

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

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

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 ::= "*" | "/"

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

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"

Parsing expressions

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)
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)])

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
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;
    else
      write m;
      m := m + 1;
    end
```
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}