D7012E Declarative Languages
Lecture 7.5

Håkan Jonsson

Luleå University of Technology

Sweden

Content

- Parsers
- Help with Lab Assignment H3
  - pages 1-9 in Lennart Andersson’s text
- (This is really Lecture 8 and we will see how far we get. It will be repeated next time at the “real” Lecture 8.)

“Parsing”

- A *parser* is a program that takes a string of characters, and produces some form of data structure that makes the syntactic structure of the string explicit
  - Parsing is all about *syntax*
  - Compilers typically parses source files
- The data structure is usually some form of *tree*
- The production is governed by a set of precise syntax rules
  - A *grammar*
- (Our) grammars are given in Backus-Naur-form (BNF)
  - A set of derivation rules expressed with
    - Meta symbols
    - Terminal symbols
    - Non-terminal symbols
- (There are different BNF:s and we will keep it simple)

BNF Example 1

Meta symbols

Non-terminal symbols

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

Terminal symbols

(Implicit meta symbol is *white-space* meaning *sequence*)

```
1 + x
```

```
4 / 3 * 2 + 1 – 0
```

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

```
4 / (3 * (2 + 1) — 0)
```

```
1 + (x * x — 2)
```


BNF Example 2

Meta symbol

Non-terminal symbol

```
match ::= match  match
match ::= ‘(‘  match  ‘)’
match ::= ‘()’
```

Terminal symbols

(What is this a description of..?)

BNF Example 3

Non-terminal symbols

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

Meta symbols

Terminal symbols


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

```
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
```

```
read n;
s := 0;
repeat
 begin
   s := s + n;
   n := n - 1;
 end
until 1-n;
write s;
```

Parser programs

- Reads ("consumes") one character at a time from left to right
  - The input is assumed to be a string
- The production rules of the grammar controls how the characters a) are read and b) are interpreted
  - (Parser can normally backtrack and "undo" the consumption of characters)
- Tries to form syntactically correct sentences and represent them by a tree
  - In our case, a value of an algebraic data type

```
"read n; s := 0; repeat begin s := s + n; n := n – …"
```

…

Parser parts

- In practice, we program *small parsers* for all basic parts in valid sentences
- We then *combine* such parsers into parser sequences to parse more complicated parts
- Example:

([],"read n; s := 0; repeat begin s := s + n; n := n – …")

([Read n], "s := 0; repeat begin s := s + n; n := n – …")

([Read "n", Assignment "s" (Num 0)], "repeat begin s := s + n; n := n – …")

…

Parser a

- Lab Assignment H3 is about parsing
- Parser operators build a parser from small parsers
  - Pretty much like how arithmetic operators (+, -, *, /) build arithmetic expressions from numerals, variables, and arithmetic expressions
- The type of a parser that parses an <u>a</u> is <u>Parser a</u> where

type Parser a = String -> Maybe (a, String)

data Maybe t = Nothing | Just t

  - The argument [of the parser] is the string from which characters are consumed
  - The result is the parsed <u>a</u> and what <u>remains of the string</u>

A small star parser

- star below is a small parser that parses a character '*' out of a string:

```
type Parser a = String -> Maybe (a, String)
```

```
star  ::  Parser Char
star ('*' : xs)  =  Just ('*', xs)
star _  =  Nothing
```

```
*Main> :type star
star :: Parser Char
*Main> star "LOL"
Nothing
*Main> star "*ling"
Just ('*',"ling")
*Main>
```

```
star "*12A"
→ Just ('*', "12A") :: Maybe (Char, String)
```

```
star "NISSE"
→ Nothing :: Maybe (Char, String)
```

star :: String -> Maybe (Char, String)

Three general small parsers

```
char :: Parser Char
char []= Nothing
char (c:cs) = Just (c, cs)
```

```
fail :: Parser a
fail cs = Nothing
```

```
return :: a -> Parser a
return x cs = Just (x, cs)
```


Parser operator no 1: Test (?)

- Combines a parser and a test of the result into a parser
  - Returns Nothing if the test on the result is false
  - Otherwise, Just with the parsed value and the remains of the string is returned

```
Infix 7 ?

(?) :: Parser a -> (a -> Bool) -> Parser a
(m ? p) cs =
   case m cs of
   Nothing -> Nothing
   Just(r, s) -> if p r then Just(r, s) else Nothing
```

```
(?) :: Parser a -> ...
(?) m p cs =
   case m cs of
   Nothin...
   Just(r, ...
```

```
char :: Parser Char
char []= Nothing
char (c:cs) = Just (c, cs)
```

+ `(=='*')` → `star2 = char ? (=='*')`

Parser operator no 2: Alternative (!)

- Combines two parsers into one
  - If the first succeeds, its result is the result
  - If the first fails, the second parser is used

```
Infix 3 !

(!) :: Parser a -> Parser a -> Parser a
(m ! n) cs = case m cs of
          Nothing -> n cs
          mcs -> mcs
```

`starOrDigit = star2 ! digit`   `lit c = char ? (==c)`   `star3 = lit '*'`

Parser operator no 3: Sequence (#)

- Applies two parsers in sequence
  - The remainder string from the first is fed into the second
  - Both results end up in a pair

```
infixl 6 #

(#) :: Parser a -> Parser b -> Parser (a, b)
(m # n) cs =
   case m cs of
   Nothing -> Nothing
   Just(a, cs') ->
      case n cs' of
      Nothing -> Nothing
      Just(b, cs'') -> Just((a, b), cs'')
```

```
twochars :: Parser (Char, Char)
twochars = char # char
```

```
twochars "abcd" → Just (('a','b'),"cd")
```

```
funcArrow =
  twochars ? (\(x,y) -> x=='-' && y=='>')
```

Parser operator no 4: Transform (>->)

- Applies a function to the result from a parser

(digitToInt can be found in Prelude)

```
infixl 5 >->

(>->) :: Parser a -> (a -> b) -> Parser b
(m >-> b) cs =
   case m cs of
   Just(a, cs') -> Just(b a, cs')
   Nothing -> Nothing
```

```
digitVal = digit >-> digitToInt
```

```
funcArrowStr = funcArrow >-> (\(x,y) -> x:y:[])
```

```
thirdchar = twochars # twochars >-> snd >-> fst
```

```
thirdchar = ( (twochars # twochars) >-> snd) >-> fst
```


Iterating parsers

```
iterate' :: Parser a -> Int -> Parser [a]
iterate' m 0 = return []
iterate' m i = m # iterate' m (i-1) >-> cons
  where
    cons (a, b) = a:b
```

```
iterate :: Parser a -> Int -> Parser [a]
iterate m 0 = return []
iterate m i =
    m # iterate m (i-1) >-> uncurry (:)
```

```
nDigits :: Int -> Parser [Int]
nDigits = iterate digitVal
```

```
iter :: Parser a -> Parser [a]
iter m = m # iter m >-> uncurry (:) ! return []
```

```
numeral' :: Parser [Char]
numeral' = iter digit
```

```
numeral:: Parser [Char]
numeral= digit # iter digit >-> uncurry (:)
```

```
iter m = ( (m # (iter m) ) >-> (uncurry (:) ) ) ! (return [])
```


Parser operator no 5: Transfer (#>)

- Transfers the result from one parser to another, so it can be further processed

```
infixl 4 #>

(#>) :: Parser a -> (a -> Parser b) -> Parser b
(p #> k) cs =
   case p cs of
   Nothing -> Nothing
   Just(a, cs') -> k a cs'
```

```
twoinrow = char #> lit
```

```
threeinrow = twoinrow #> lit
```