Script generated by TTT

Title: Nipkow: Info2 (30.01.2015)

Date: Fri Jan 30 08:30:32 CET 2015

Duration: 58:50 min

Pages: 82

15. Case Study: Parsing

15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree

15.1 Basic Parsing

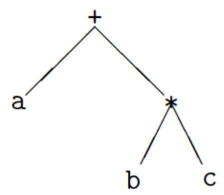
Parsing is the translation of a string into a syntax tree according to some grammar.

15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

"a+b*c" $\mapsto$



Parser type

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

Parser type

```
type Parser = String -> Tree

type Parser a = String -> a
```

Parser type

```
type Parser = String -> Tree

type Parser a = String -> a
```

What if something is left over, e.g., "a+b*c#" ?

Parser type

`type Parser = String -> Tree`

`type Parser a = String -> a`

What if something is left over, e.g., `"a+b*c#"` ?

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

Parser type

`type Parser = String -> Tree`

`type Parser a = String -> a`

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

Parser type

`type Parser = String -> Tree`

`type Parser a = String -> a`

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

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

Parser type

`type Parser = String -> Tree`

`type Parser a = String -> a`

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

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

`[]` syntax error

Parser type

```
type Parser = String -> Tree

type Parser a = String -> a
```

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

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

`[]` syntax error

`[x]` one result x

Parser type

```
type Parser = String -> Tree

type Parser a = String -> a
```

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

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

`[]` syntax error

`[x]` one result x

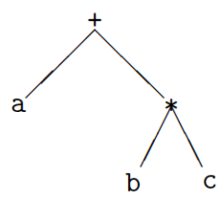
`[x,y,...]` multiple results, ambiguous language

15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

`"a+b*c"` $\mapsto$



Parser type

```
type Parser = String -> Tree

type Parser a = String -> a
```

What if something is left over, e.g., `"a+b*c#"` ?

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

What if there is a syntax error, e.g., `"++"` ?

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

`[]` syntax error

`[x]` one result x

`[x,y,...]` multiple results, ambiguous language

Alternative parser type

For unambiguous languages:

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

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)
```

Example

```
char 'a' "abc" =
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)
```

Example

```
char 'a' "abc" = [('a',"bc")]
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)
```

Example

```
char 'a' "abc" = [('a',"bc")]
char 'b' "abc" = []
```

Combining parsers

Parse anything that p1 or p2 can parse:

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

Combining parsers

Parse anything that p1 or p2 can parse:

```
(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2  =
```

Combining parsers

Parse anything that p1 or p2 can parse:

```
(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2  =  \cs -> p1 cs ++ p2 cs
```

Example

```
(char 'b' ||| char 'a') "abc" =
```

Combining parsers

Parse anything that p1 or p2 can parse:

```
(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2  =  \cs -> p1 cs ++ p2 cs
```

Example

```
(char 'b' ||| char 'a') "abc" =
```

Combining parsers

Parse anything that p1 or p2 can parse:

```
(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2  =  \cs -> p1 cs ++ p2 cs
```

Example

```
(char 'b' ||| char 'a') "abc" = [('a',"bc")]
```

Combining parsers

Parse first with p1, then the remainder with p2:

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

Combining parsers

Parse first with p1, then the remainder with p2:

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


```
(p1 *** p2) xs =
```

Combining parsers

Parse first with p1, then the remainder with p2:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```


Combining parsers

Parse first with p1, then the remainder with p2:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```


Example

```
(char 'b' *** char 'a') "bac" =
```


Combining parsers

Parse first with p1, then the remainder with p2:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```

Example


```
(char 'b' *** char 'a') "bac" = [(('a','b'), "c")]
```


Combining parsers

Parse first with p1, then the remainder with p2:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```

Example

```
(char 'b' *** char 'a') "bac" = [(('a','b'), "c")]
```


```
(one isAlpha *** one isDigit *** one isDigit) "a12"
```

Combining parsers

Parse first with p1, then the remainder with p2:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```

Example

```
(char ’b’ *** char ’a’) "bac" = [((’a’,’b’), "c")]

(one isAlpha *** one isDigit *** one isDigit) "a12"
= [((’a’,(’1’,’2’)), "")]
```

Transforming the result

Parse with p, transform result with f:

```
(>>>) :: Parser a -> (a -> b) -> Parser b
p >>> f  =
```

Transforming the result

Parse with p, transform result with f:

```
(>>>) :: Parser a -> (a -> b) -> Parser b
p >>> f  =  \xs -> [(f a,ys) | (a,ys) <- p xs]
```

Transforming the result

Parse with p, transform result with f:

```
(>>>) :: Parser a -> (a -> b) -> Parser b
p >>> f  =  \xs -> [(f a,ys) | (a,ys) <- p xs]
```

Example

```
((char ’b’ *** char ’a’) >>> (\(x,y) -> [x,y])) "bac"
```

Transforming the result

Parse with `p`, transform result with `f`:

```
(>>>) :: Parser a -> (a -> b) -> Parser b
p >>> f  =  \xs -> [(f a,ys) | (a,ys) <- p xs]
```

Example

```
((char 'b' *** char 'a') >>> (\(x,y) -> [x,y])) "bac"
= [(("ab"), "c")]
```

Parsing a list of objects

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b
```

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =
```


Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:


```
list :: Parser a -> Parser [a]
list p  =  (p *** list p)
```


Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:


```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||
```


Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:


```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||   success []
```

```
Terminal Shell Edit View Window Help                Fri 09:09
Code — ghc — 71×24
2013/te
xmf-dist/fonts/type1/public/amsfonts/cm/cmtt8.pfb></usr/local/texlive/2
)13/texm
f-dist/fonts/type1/public/amsfonts/latxfont/line10.pfb></usr/local/texl
ive/2013
/texmf-dist/fonts/type1/public/amsfonts/symbols/msam10.pfb></usr/local/
texlive/
2013/texmf-dist/fonts/type1/public/amsfonts/symbols/msbm10.pfb>
Output written on slides.pdf (445 pages, 994316 bytes).
Transcript written on slides.log.
lapnipkow1d:Slides nipkow$ open slides.pdf
lapnipkow1d:Slides nipkow$ cp-slides
slides.pdf                              100%  971KB 971.0KB/s   0
0:00
lapnipkow1d:Slides nipkow$ cd ../Code/
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Parser
[1 of 1] Compiling Parser           ( Parser.hs, interpreted )
Ok, modules loaded: Parser.
*Parser> list (one isAlpha) "ab1"_
```

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||   success []
```

```
Terminal Shell Edit View Window Help                Fri 09:11
Code — ghc — 66×23
/texmf-dist/fonts/type1/public/amsfonts/symbols/msam10.pfb></u
sr/local/texlive/
2013/texmf-dist/fonts/type1/public/amsfonts/symbols/msbm10.pfb
>
Output written on slides.pdf (445 pages, 994316 bytes).
Transcript written on slides.log.
lapnipkow1d:Slides nipkow$ open slides.pdf
lapnipkow1d:Slides nipkow$ cp-slides
slides.pdf                              100%  971KB 971.
0KB/s   00:00
lapnipkow1d:Slides nipkow$ cd ../Code/
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Parser
[1 of 1] Compiling Parser           ( Parser.hs, interpreted )
Ok, modules loaded: Parser.
*Parser> list (one isAlpha) "ab1"
[("ab","1"),("a","b1"),("","ab1")]
*Parser> _
```


Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||   success []
```

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||   success []
```

Example

```
list (one isAlpha) "abc123" =
```


Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||   success []
```

Example


```
list (one isAlpha) "abc123" = [("abc", "123")]
```

Parsing a non-empty list of objects

```
list1 :: Parser a -> Parser [a]
list1 p  =  (p *** list p) >>> uncurry (:)
```

Parsing identifiers

```
ident :: Parser String
```

Parsing identifiers

```
ident :: Parser String
ident  =  (list1(one isAlpha) *** list(one isDigit))
```

Parsing identifiers

```
ident :: Parser String
ident  =  (list1(one isAlpha) *** list(one isDigit))
          >>> uncurry (++)
```

Parsing identifiers

```
ident :: Parser String
ident  =  (list1(one isAlpha) *** list(one isDigit))
          >>> uncurry (++)
```

Example

```
ident "abc0++" =
```

Parsing identifiers

```
ident :: Parser String
ident  =  (list1(one isAlpha) *** list(one isDigit))
          >>> uncurry (++)
```

Example

```
ident "abc0++" = [("abc0", "++")]
```

Handling spaces

```
spaces :: Parser String
```

Handling spaces

```
spaces :: Parser String
spaces  =  list (one isSpace)

sp :: Parser a -> Parser a
```

Handling spaces

```
spaces :: Parser String
spaces  =  list (one isSpace)

sp :: Parser a -> Parser a
sp p  =  (spaces *** p)
```

Handling spaces

```
spaces :: Parser String
spaces  =  list (one isSpace)

sp :: Parser a -> Parser a
sp p  =  (spaces *** p) >>> snd
```

Example

```
(sp ident) " abc d" =
```

Handling spaces

```
spaces :: Parser String
spaces  =  list (one isSpace)

sp :: Parser a -> Parser a
sp p  =  (spaces *** p) >>> snd
```

Example

```
(sp ident) " abc d" = [("abc", " d")]
```

15.2 Application: Parsing pico-Haskell expressions

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{rcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ .\ \mathit{expr}\ ) \end{array}$$

Examples `a, (f x)`

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{rcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ .\ \mathit{expr}\ ) \end{array}$$

Examples `a, (f x), (\x. (f x))`

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ .\ \mathit{expr}\ ) \end{array}$$

Examples `a, (f x), (\x. (f x))`

The tree representation:

```
data Expr = Id String | App Expr Expr | Lam String Expr
```

Examples
```
Id "a"
```

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ .\ \mathit{expr}\ ) \end{array}$$

Examples `a, (f x), (\x. (f x))`

The tree representation:

```
data Expr = Id String | App Expr Expr | Lam String Expr
```

Examples
```
Id "a"
App (Id "f") (Id "x")
```

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ .\ \mathit{expr}\ ) \end{array}$$

Examples `a, (f x), (\x. (f x))`

The tree representation:

```
data Expr = Id String | App Expr Expr | Lam String Expr
```

Examples
```
Id "a"
App (Id "f") (Id "x")
Lam "x" (App (Id "f") (Id "x"))
```


Pico-Haskell parser

```
ch c  =  sp (char c)
id    =  sp ident

expr  =
```

```
ch c  =  sp (char c)
id    =  sp ident

expr  =
```

Pico-Haskell parser

```
ch c  =  sp (char c)
id    =  sp ident

expr  =
 id >>> Id
```

Pico-Haskell parser

```
ch c  =  sp (char c)
id    =  sp ident

expr  =
 id >>> Id
 |||
 (ch '(' *** expr *** expr *** ch ')')
   >>> (\(_,(e1,(e2,_))) -> App e1 e2)
 |||
 (ch '(' *** ch '\' *** id *** ch '.' *** expr *** ch ')')
   >>> (\(_,(_,(x,(_,(e,_))))) -> Lam x e)
```

15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$
\begin{array}{rcl}
\textit{expr} & ::= & \textit{identifier} \\
& | & (\ \textit{expr}\ \textit{expr}\ ) \\
& | & (\ \backslash\ \textit{identifier}\ .\ \textit{expr}\ )
\end{array}
$$

Examples `a`, `(f x)`, `(\x. (f x))`

The tree representation:

```
data Expr = Id String | App Expr Expr | Lam String Expr
```

Examples
```
Id "a"
App (Id "f") (Id "x")
Lam "x" (App (Id "f") (Id "x"))
```

15.3 Improved Parsing

`String` $\xrightarrow{\text{Lexer}}$ `[Token]` $\xrightarrow{\text{Parser}}$ `Tree`

Example

```
data Token =
  LParant | RParant | BSlash | Dot | Ident String
```

15.3 Improved Parsing

`String` $\xrightarrow{\text{Lexer}}$ `[Token]` $\xrightarrow{\text{Parser}}$ `Tree`

Example

```
data Token =
  LParant | RParant | BSlash | Dot | Ident String
```

`"(\x1 . x2)"` $\xrightarrow{\text{Lexer}}$

```
  [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]
```

Why?

15.3 Improved Parsing

`String` $\xrightarrow{\text{Lexer}}$ `[Token]` $\xrightarrow{\text{Parser}}$ `Tree`

Example

```
data Token =
  LParant | RParant | BSlash | Dot | Ident String
```

`"(\x1 . x2)"` $\xrightarrow{\text{Lexer}}$

```
  [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]
```

Why?

- Lexer based on regular expressions
  $\Longrightarrow$ lexer can be more efficient than general parser

Generalizing the implementation

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

15.3 Improved Parsing

$$\texttt{String} \xrightarrow{\text{Lexer}} \texttt{[Token]} \xrightarrow{\text{Parser}} \texttt{Tree}$$

Example

```
data Token =
  LParant | RParant | BSlash | Dot | Ident String
```

`"(\x1 . x2)"` $\xrightarrow{\text{Lexer}}$

```
  [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]
```

Why?

- Lexer based on regular expressions
  $\Longrightarrow$ lexer can be more efficient than general parser
- Lexer can already remove spaces and comments
  $\Longrightarrow$ simplifies parsing

Generalizing the implementation

So far:

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

Now:

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

None of the parser combinators ***, |||, >>> change,
only their types become more general!

So far:

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

Now:

```
(***) ::
```

Some literature:

- Chapter 8 of Hutton's *Programming in Haskell*
- Section 17.5 in Thompson's Haskell book (3rd edition)
- Many papers on functional parsers