MOD08: Functional Programming

Parsing in Haskell

Marco Gerards

03-05-2019

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
5-7	- Project

Organisation of the course

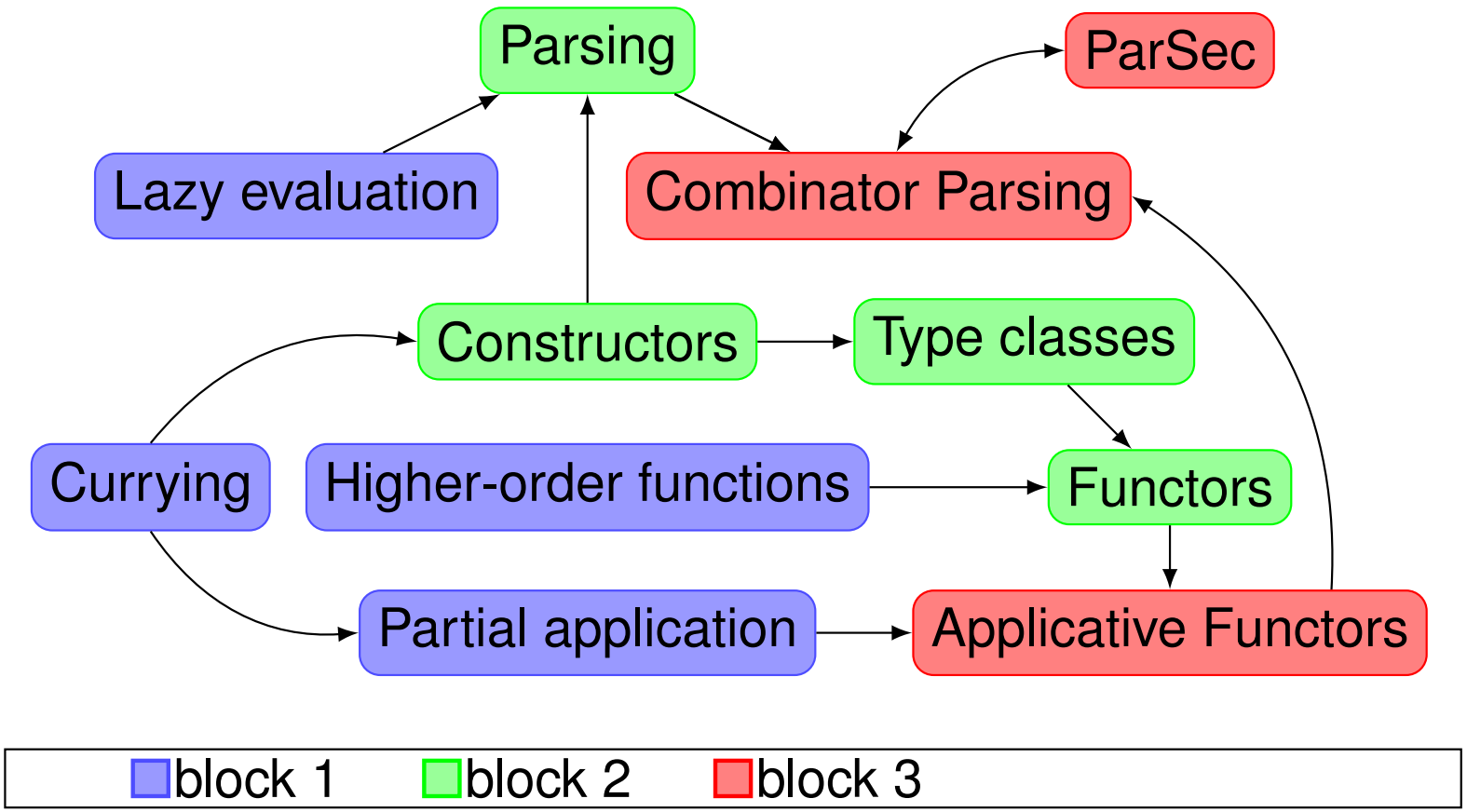
Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
5-7	- Project

This lecture: learning goals

"Solve non-trivial programming problems in Functional Programming"

Application: Parsing

Connection of some of the topics between blocks



Goal of the lecture

Input:

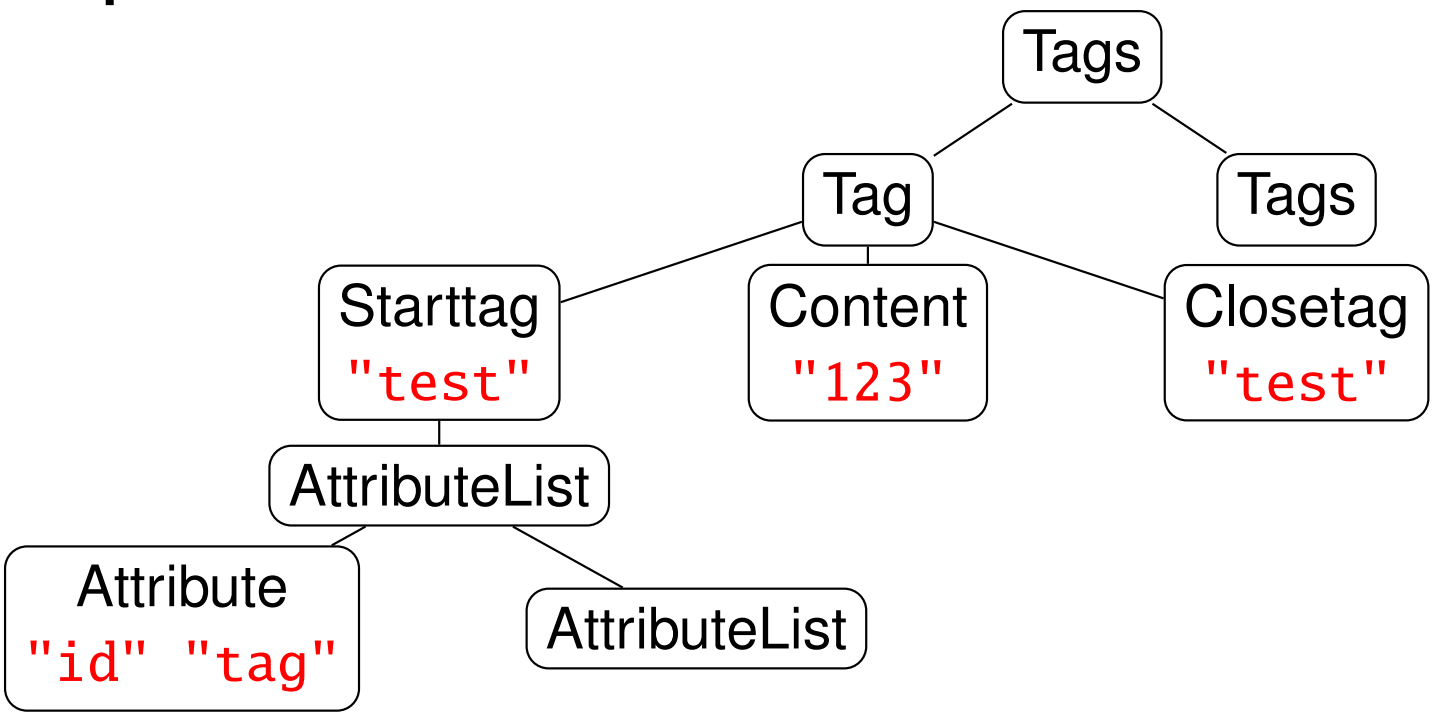
```
<test id="tag">123</test>
```

Goal of the lecture

Input:

```
<test id="tag">123</test>
```

Output:



Tokenizing

Input: `<test id="tag">123</test>`

Output:

```
[LBrack, Id "test", Id "id", Assignment,
 ConstStr "tag", RBrack,Id "123", Close,
 Id "test", RBrack]
```

Tokenizer

- Convert a string to a list of tokens
- Synonyms: lexer, scanner

Tokenizer

- Convert a string to a list of tokens
- Synonyms: lexer, scanner
- Example:
  - Input tokenizer:

    ```
    int i = *p + 1024;
    ```

  - Output tokenizer:

    ```
    [ID "int", ID "i", IS, STAR "*",
     ID "p", PLUS, CONST 1024, SEMIC]
    ```

Tokenizer

- Convert a string to a list of tokens
- Synonyms: lexer, scanner
- Example:
  - Input tokenizer:

    ```
    int i = *p + 1024;
    ```

  - Output tokenizer:

    ```
    [ID "int", ID "i", IS, STAR "*",
     ID "p", PLUS, CONST 1024, SEMIC]
    ```

- This lecture: simple (ad-hoc) tokenizer
  - Normally: FSA
  - Ad-hoc solution; later ParSec

Implementation: Token type

```
data Token = LBrack             -- '<'
           | RBrack             -- '>'
           | Close              -- '</'
           | Assignment         -- '='
           | ConstStr String    -- " ... "
           | Id String          -- [a-zA-Z][a-zA-Z0-9]+
          deriving (Show, Eq)
```

Implementation: simple tokenizer (1/2)

```
tokenize :: String -> [Token]
tokenize (' ':xs)          = tokenize xs
tokenize ('\t':xs)         = tokenize xs
tokenize ('<': '/' : xs)   = Close      : tokenize xs
tokenize ('/': '>' : xs)   = EClose     : tokenize xs
tokenize ('<' : xs)        = LBrack     : tokenize xs
tokenize ('>' : xs)        = RBrack     : tokenize xs
tokenize ('=' : xs)        = Assignment : tokenize xs
                    ...
```

Implementation: simple tokenizer (2/2)

```
                    ...

tokenize ('=':xs)         = Assignment : tokenize xs
tokenize ('"':xs)         = ConstStr string : tokenize rem
   where (string,'"':rem) = break (=='"') xs
tokenize xs               = Id id : tokenize rem
   where (id, rem) = span (`elem` letdig) xs


letter = ['a'..'z'] ++ ['A'..'Z']
digit  = ['0' .. '9']
letdig = letter ++ digit
```

Parsing: Context-Free Grammars

- **Productions:**

$$S \Rightarrow a\ S\ b$$

$$S \Rightarrow c\ R$$

$$R \Rightarrow d\ S\ d$$

$$R \Rightarrow e$$

  - **Nonterminals:** S , R
  - **Terminals (tokens):** a , b and c
  - **Start symbol:** S

Parsing: Context-Free Grammars

- **Productions:**

$$S \Rightarrow a\ S\ b$$
$$S \Rightarrow c\ R$$
$$R \Rightarrow d\ S\ d$$
$$R \Rightarrow e$$

  - **Nonterminals:** S, R
  - **Terminals (tokens):** a , b and c
  - **Start symbol:** S

Parsing: Context-Free Grammars

- **Productions:**

S	$\Rightarrow$	$a\ S\ b$
S	$\Rightarrow$	$c\ R$
R	$\Rightarrow$	$d\ S\ d$
R	$\Rightarrow$	e

  - **Nonterminals:** S, R
  - **Terminals (tokens):** a, b and c
  - **Start symbol:** S

Parsing: Context-Free Grammars

- **Productions:**

S	$\Rightarrow$	$a\ S\ b$
S	$\Rightarrow$	$c\ R$
R	$\Rightarrow$	$d\ S\ d$
R	$\Rightarrow$	e

  - **Nonterminals:** S, R
  - **Terminals (tokens):** a, b and c
  - **Start symbol:** S
- Language
  - Strings that can be produced using the productions
  - Example: *ce*, *aaacebbb*, *cdced*

Backus Naur Form (BNF)

Productions:

$$
\begin{array}{lcl}
S & \Rightarrow & a\ S\ b \\
S & \Rightarrow & c\ R \\
R & \Rightarrow & d\ S\ d \\
R & \Rightarrow & e
\end{array}
$$

BNF:

```
<S>     ::= a <S> b
          |  c <R>
<R>     ::= d <S> d
          |  e
```

Backus Naur Form (BNF)

Productions:

$S \Rightarrow a\ S\ b$

$S \Rightarrow c\ R$

$R \Rightarrow d\ S\ d$

$R \Rightarrow e$

BNF:

```
<S>     ::= a <S> b
          |   c <R>
<R>     ::= d <S> d
          |   e
```

Backus Naur Form (BNF)

Productions:

S	$\Rightarrow$	$a\ S\ b$
S	$\Rightarrow$	$c\ R$
R	$\Rightarrow$	$d\ S\ d$
R	$\Rightarrow$	e

BNF:

```
<S>     ::= a <S> b
          |   c <R>
<R>     ::= d <S> d
          |   e
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input:

```
< test name = "tag"  >
```

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input:

```
< test name = "tag"  >
```

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>       ::= '<' tagid <attributelist> '>'
<attributelist>  ::= <attribute> <attributelist>
                   | epsilon
<attribute>      ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

Parsing: Grammars (example)

Input: `< test name = "tag" >`

Tokens:

```
[LBrack,Id "test",Id "name", Is, Str "tag", RBrack]
```

Grammar:

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

What does this code do?

```
g 0 = 42
h x = 42 / x

f x | x == 0    = p
    | otherwise = q
  where p = g x
        q = h x
```

What does this code do?

```
g 0 = 42
h x = 42 / x

f x | x == 0    = p
    | otherwise = q
  where p = g x
        q = h x
```

- What is `f 42`?

What does this code do?

```
g 0 = 42
h x = 42 / x

f x | x == 0    = p
    | otherwise = q
  where p = g x
        q = h x
```

- What is `f 42`?
- What is `f 0`?

What does this code do?

```
g 0 = 42
h x = 42 / x

f x | x == 0    = p
    | otherwise = q
  where p = g x
        q = h x
```

- What is `f 42`?
- What is `f 0`?

Lazy evaluation

Recursive descent parsing

- Each nonterminal becomes a function
- No backtracking; one token look-ahead
- Tree
  - Parse tree (corresponds to grammar)
  - Abstract syntax tree

Example of simple recursive descent parser (1/4)

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

General definitions:

```
la :: [Char] -> [Char] -> Bool -- look-ahead of one token
la xs l = length xs > 0 && (head xs) `elem` l

data NT    = S | Q |R
           deriving Show
data ParseTree = Leaf Char
               | Node NT [ParseTree]
           deriving Show
```

Example of simple recursive descent parser (2/4)

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Parser for S:

```
parseS :: [Char] -> (ParseTree, [Char])
parseS xs | la xs "a"  = (Node S [tq], ys) -- Q
          | la xs "bc" = (Node S [tr], zs) -- R
          | otherwise  = error "parse error in S"
  where (tq, ys) = parseQ xs
        (tr, zs) = parseR xs
```

Example of simple recursive descent parser (2/4)

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Parser for S:

```
parseS :: [Char] -> (ParseTree, [Char])
parseS xs | la xs "a"  = (Node S [tq], ys) -- Q
          | la xs "bc" = (Node S [tr], zs) -- R
          | otherwise  = error "parse error in S"
  where (tq, ys) = parseQ xs
        (tr, zs) = parseR xs
```

Lazy evaluation: either `parseQ` *or* `parseR` is used

Example of simple recursive descent parser (3/4)

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Parser for Q:

```
parseQ :: [Char] -> (ParseTree, [Char])
parseQ ('a':'b':xs) = (Node Q [Leaf 'a', Leaf 'b'], xs)
parseQ xs = error "parse error in Q"
```

Example of simple recursive descent parser (4/4)

```
<S> ::= <Q> | <R>
<Q> ::= 'a' 'b'
<R> ::= 'b' <Q> <Q>
      | 'c' <Q> 'c'
```

Parser for R:

```
parseR :: [Char] -> (ParseTree, [Char])
parseR ('b':xs) = (Node R [Leaf 'b', t1, t2], zs)
 where (t1, ys) = parseQ xs -- 'b' Q
       (t2, zs) = parseQ ys -- 'b' Q Q
parseR ('c':xs) = (Node R [Leaf 'b', t1, Leaf 'c'], zs)
 where (t1, ys) = parseQ xs -- 'c' Q
       ('c':zs) = ys        -- 'c' Q 'c'
```

HTML Tag Parser in Haskell: Tokens

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

```haskell
data Token = LBrack              -- '<'
           | RBrack              -- '>'
           | Assignment          -- '='
           | ConstStr String     -- " ... "
           | Id String           -- [a-zA-Z]+
           deriving (Show, Eq)
```

HTML Tag Parser in Haskell: Tree

```
<starttag>      ::= '<' tagid <attributelist> '>'
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
<attribute>     ::= id '=' str
```

```
data NT = NTST | NTAttributeList | NTAttribute
        deriving Show

data ParseTree = Leaf Token
               | Node NT [ ParseTree ]
               deriving Show
```

HTML Tag Parser in Haskell: attribute

```
<attribute>     ::= id '=' str
```

```
parseAttr :: [Token] -> (ParseTree, [Token])
parseAttr (tid@(Id id):Assignment:tc@(ConstStr str):xs)
  = (Node NTAttribute [Leaf tid,
                       Leaf Assignment,
                       Leaf tc],
     xs)
```

```
*Main> tokenize "attr=\"test\""
[Id "attr",Assignment,ConstStr "test"]
*Main> parseAttr [Id "attr",Assignment,ConstStr
"test"]
(Leaf (NTAttribute "attr" "test"),[])
```

HTML Tag Parser in Haskell: attributelist

```
<attributelist> ::= <attribute> <attributelist>
                  | epsilon
```

```
parseAttrL :: [Token] -> (ParseTree, [Token])
parseAttrL xs@((Id _):_) = (Node NTAttributeList [tattr,
                                                  ttail],
                            zs)
  where (tattr, ys) = parseAttr xs
        (ttail, zs) = parseAttrL ys
parseAttrL  xs = (Node NTAttributeList [], xs) -- epsilon
```

HTML Tag Parser in Haskell: starttag

```
<starttag>      ::= '<' tagid <attributelist> '>'
```

```
parseST :: [Token] -> (ParseTree, [Token])
parseST (LBrack : id@(Id tag) : xs) = (Node NTStartTag
                                        [Leaf LBrack,
                                         Leaf id,
                                         tattr,
                                         Leaf RBrack],
                                       ys)
  where (tattr, RBrack: ys) = parseAttrL xs
```

Example: EBNF (required for practical session)

```
<A> ::= <B> (<B> | <C>)
<B> ::= 'b'
<C> ::= 'c'
```

Example: EBNF (required for practical session)

```
<A> ::= <B> (<B> | <C>)
<B> ::= 'b'
<C> ::= 'c'
```

Parser:

```
parseA :: [Char] -> (ParseTree, [Char])
parseA xs | la ys "b" = (Node A [t1, t2 ], zs)
          | la ys "c" = (Node A [t1, t2'], zs')
  where (t1,  ys)  = parseB xs -- B
        (t2,  zs)  = parseB ys -- B B
        (t2', zs') = parseC ys -- B C

parseB ('b':xs) = (Node B [Leaf 'b'], xs)
parseC ('c':xs) = (Node C [Leaf 'c'], xs)
```

Parse trees vs ASTs (1/2)

```
<expr>   ::=  <term> "+" <expr>
          |    <term>
<term>   ::=  <factor> "*" <term>
          |    <factor>
<factor> ::= num
          |   "(" <expr> ")"
```

Parse trees vs ASTs (1/2)

```
<expr>   ::=  <term> "+" <expr>
          |    <term>
<term>   ::=  <factor> "*" <term>
          |    <factor>
<factor> ::= num
          |   "(" <expr> ")"
```

Expression	Parse tree	AST
4	<expr> | <term> | <factor> | 4	4

Parse trees vs ASTs (2/2)

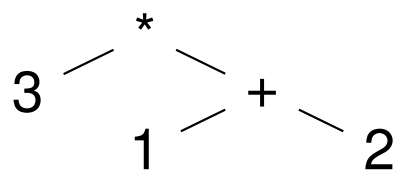
```
<expr>   ::=  <term> "+" <expr>
          |    <term>
<term>   ::=  <factor> "*" <term>
          |    <factor>
<factor> ::= num
          |   "(" <expr> ")"
```

- AST for $3 * (1 + 2)$?

Parse trees vs ASTs (2/2)

```
<expr>   ::=  <term> "+" <expr>
           |   <term>
<term>   ::=  <factor> "*" <term>
           |   <factor>
<factor> ::= num
           |   "(" <expr> ")"
```

- AST for $3 * (1 + 2)$?
- AST:



Some hints for the practical session

- Use ad-hoc tokenizer based on these slides (you may copy from slides!)

Some hints for the practical session

- Use ad-hoc tokenizer based on these slides (you may copy from slides!)
- Use an AST: BinTree is sufficient (see last slides)

Some hints for the practical session

- Use ad-hoc tokenizer based on these slides (you may copy from slides!)
- Use an AST: BinTree is sufficient (see last slides)
- Grammar is not LL(1); study `parseA` to get get started

Some hints for the practical session

- Use ad-hoc tokenizer based on these slides (you may copy from slides!)
- Use an AST: BinTree is sufficient (see last slides)
- Grammar is not LL(1); study `parseA` to get get started
- Types of parsers (for some `a` and `b` determined by you):
  - String parser:
    ```
    parse... :: String -> (BinTree a b, String)
    ```
  - Token parser:
    ```
    parse... :: [Token] -> (BinTree a b, [Token])
    ```

Self study

- This lecture: “Parsing in Haskell”
  - Recursive descent parsing
  - Important for lab and the FP project; no exam material
  - MOD08 students: part of compiler construction
  - Other students: self-study; suggestions:
    - “CS143 Notes: Parsing”, David L. Dill
    - Compiler construction video lectures (Canvas)

Self study

- This lecture: “Parsing in Haskell”
  - Recursive descent parsing
  - Important for lab and the FP project; no exam material
  - MOD08 students: part of compiler construction
  - Other students: self-study; suggestions:
    - “CS143 Notes: Parsing”, David L. Dill
    - Compiler construction video lectures (Canvas)

- Next lecture (09-05): “Combinator parsing with ParSec”
  - Required background knowledge:
    - Functors
    - Partial application / currying
  - Books (do not read yet):
    - Programming in Haskell (Hutton): Chapter 13
    - ParSec: Real World Haskell: Chapter 16
  - Important for lab and the FP project; no exam material