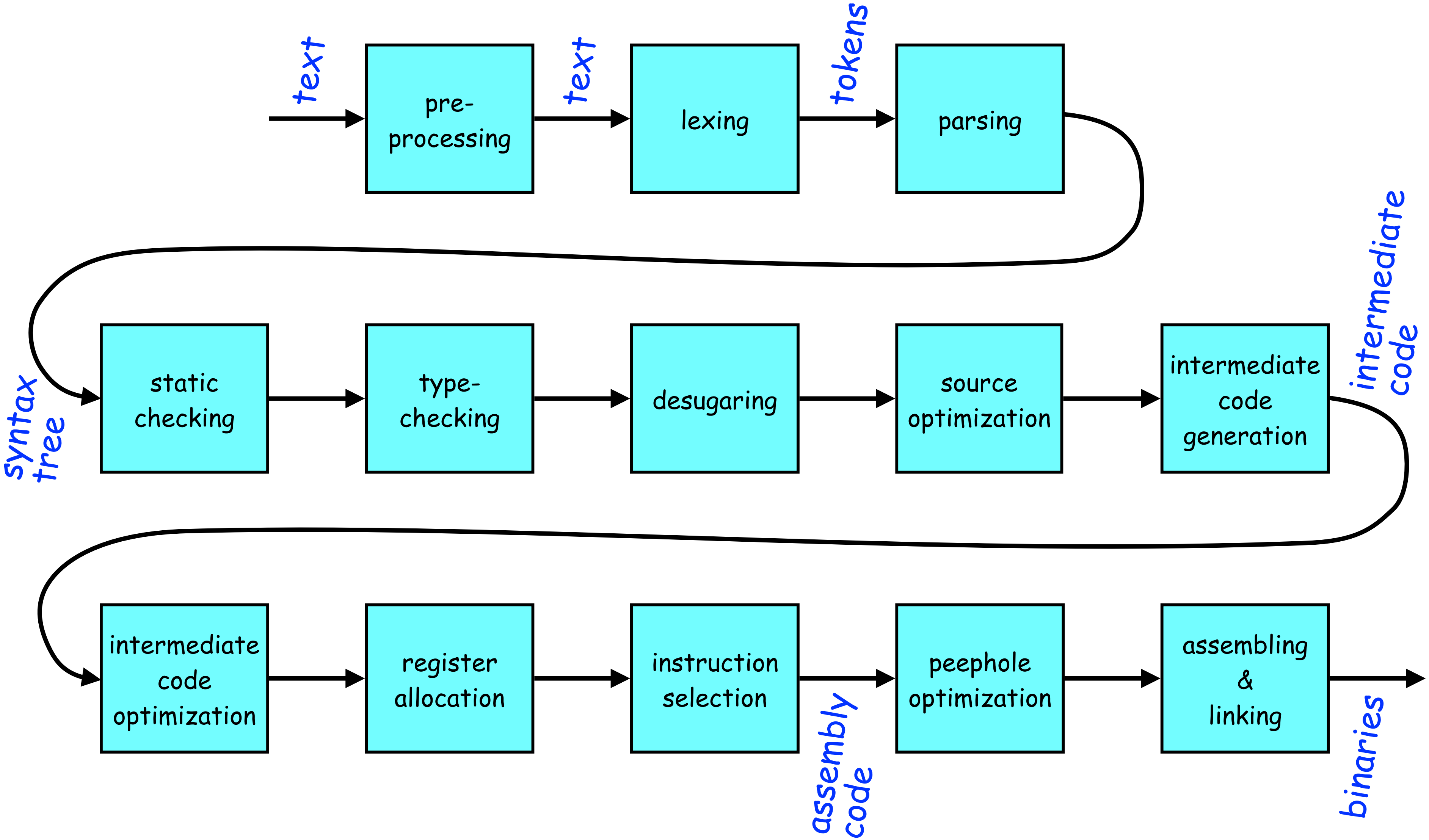
Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

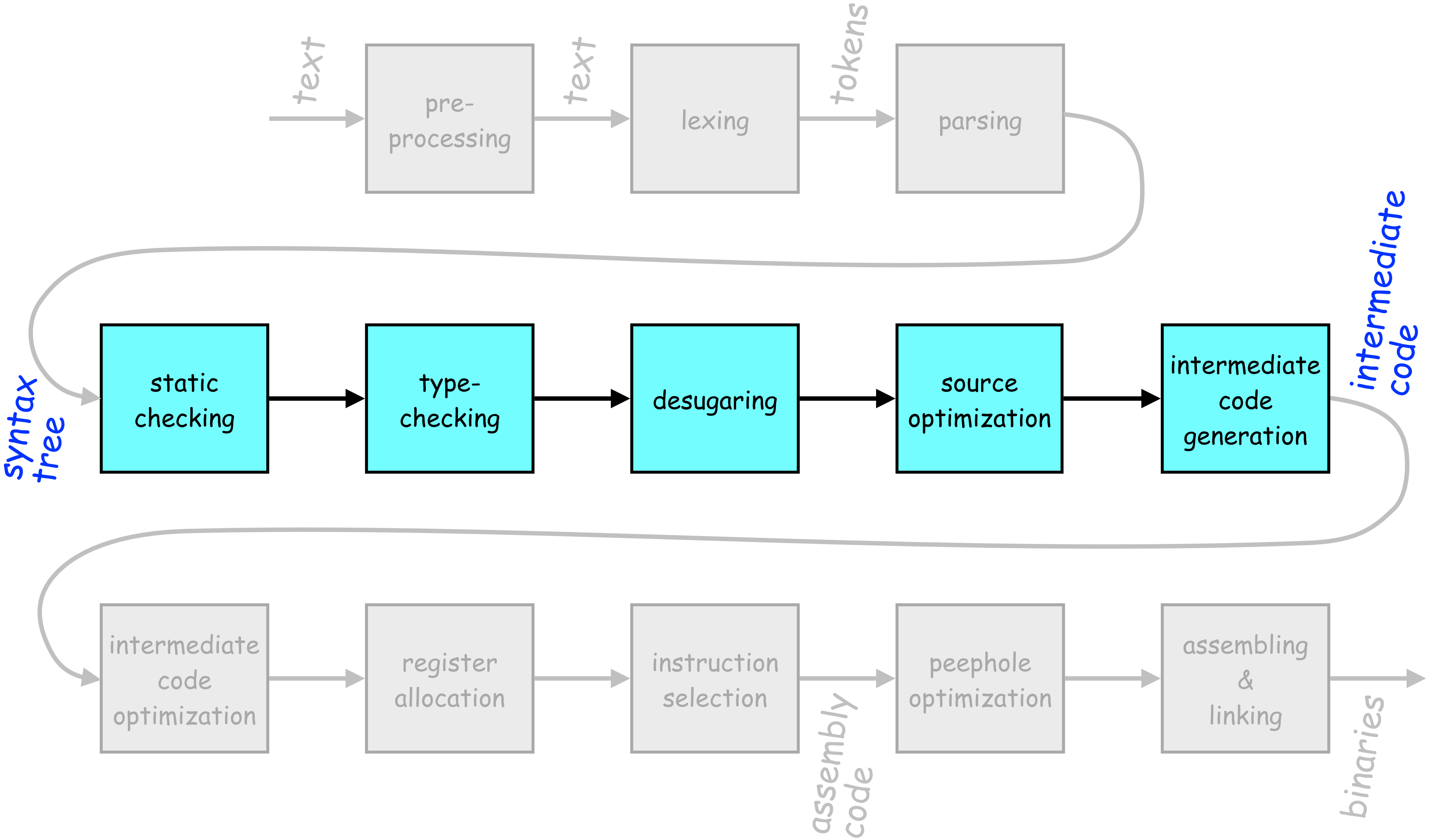
Lecture 1
Source-to-source transformations

Johan Nordlander

The compiler pipeline



This course


text
pre-processing
text
lexing
tokens
parsing
syntax tree
static checking
type-checking
desugaring
source optimization
intermediate code generation
intermediate code
intermediate code optimization
register allocation
instruction selection
assembly code
peephole optimization
assembling & linking
binaries

This course

- Techniques for analyzing & transforming a functional language, where
  - input is a correctly parsed syntax tree
  - output is intermediate code in C syntax
- Rationale: front-end (lexing, parsing) and back-end (register allocation, etc) issues not so specific to functional languages

Our functional language

- Fictional
- Pure
- Strongly typed
- Haskell-like
- Strict!
- Open to both restriction & extension

Why strict?

- To focus on other issues besides laziness
- To demonstrate similarities between functional & traditional program execution
- To enable use of plain *C* as a back-end
- Still: laziness <u>will</u> be covered, but towards the end of the course

Course overview

- Seven lectures
- Two paper presentations per student
- An individual lab project
- Examination (7.5 hp):
  - Satisfactory presentation of own papers
  - Participation in <u>all</u> paper presentations
  - Oral lab project demo
  - Written lab project report

Lecture plan

- Source-to-source transformations
- C representation
- Memory management
- Type inference
- Haskell-style overloading
- Type-based optimization
- Lazy evaluation

week 12

week 15

week 19

Paper topics

- Alternative data representations
- Advanced memory management
- Additional transformations
- Type system variations
- Efficient state and IO monads
- Parallel execution
- ...
- Your choice!

Presentations during week 19

Lab project

- Implement a compiler for your functional language (= a flavor of "our" language)
- Implementation language: your choice (but Haskell is recommended)
- Back-end: your favorite C compiler
- Front-end recommendation: the haskell-src or haskell-src-exts packages

A common theme

- Manipulation of syntax trees — schematically:

Input:

```
parse :: String -> SyntaxTree
```

Verification / addition of missing information:

```
staticCheck :: SyntaxTree -> Bool
typeInference :: SyntaxTree -> SyntaxTree
```

Misc. transformations, possibly changing representation:

```
desugar :: SyntaxTree -> SyntaxTree
translate :: SyntaxTree -> CoreSyntaxTree
optimize :: CoreSyntaxTree -> CoreSyntaxTree
```

Output:

```
codegen :: CoreSyntaxTree -> String
```

Source-to-source transformations

- Rewriting syntax trees with the purpose of
  - Removing redundant constructs today
  - Making implicit information explicit
  - Choosing more efficient representations
  - Normalizing form before code generation
- Can be distributed over different passes, run in many different orders

In haskell-src

```
data HsModule     =  HsModule SrcLoc Module ... [HsDecl]
data HsDecl       =  HsTypeDecl SrcLoc HsName [HsName] HsType
                  |  HsFunBind [HsMatch]
                  |  ...
data HsMatch      =  HsMatch SrcLoc HsName [HsPat] HsRhs [HsDecl]
data HsExp        =  HsVar HsQName
                  |  HsCon HsLiteral
                  |  HsApp HsExp HsExp
                  |  HsLambda SrcLoc [HsPat] HsExp
                  |  HsListComp HsExp [HsStmt]
                  |  HsRightSection HsQOp HsExp
                  |  ...
data HsQOp        =  HsQVarOp HsQName | HsQConOp HsQName
data HsQName      =  Qual Module HsName | UnQual HsName | ...
...
```

A transformation

- Removing operator sections:

```
...
translate nameSupply (HsRightSection op e)  =
        HsLambda nullSrcLoc [HsPVar x]
             HsApp (HsApp (opToExp op) (HsVar (UnQual x)))
                       (translate nameSupply' e)
  where (nameSupply', x) = newName nameSupply
...

opToExp (HsQVarOp qname) = HsVar qname
opToExp (HsQConOp qname) = HsCon qname
```

A transformation

- Removing operator sections using "concrete" abstract syntax:

```
...
translate  (op e)       =      \x -> x op e
    where x is a new variable
...
```

Concrete abstract syntax

- Written in blue, meta-syntax in black
- Represents trees – no ambiguity worries!
- Certain variables denote arbitrary subtrees (e for expressions; x,y,z for names; etc)
- Plural suffix s denotes lists (as in es)
- Mix with list meta-syntax ([], e:es, e,es, es++es')
- Indexing and ellipsis: [e_1, ..., e_n]
- Relaxed patterns (e.g. non-linear, or es_1++...++es_n)

Our input language

```
prog  ::=  module K where ds
d     ::=  p rhs  |  ms  |  ...
m     ::=  x ps rhs  |  x ps rhs where ds
rhs   ::=  = e  |  grhss
grhs  ::=  | e = e
e     ::=  x  |  K  |  lit  |  e op e  |  e e  |  - e  |  \ps -> e  |  let ds in e  |
           if e then e else e  |  case e of alts  |  (es)  |  [es]  |  [e..e]  |
           [e,e..e]  |  [ e | stmts ]  |  (e op)  |  (op e)  | K { fs }  |  e { fs }
p     ::=  x  |  K ps  |  lit  |  - p  |  p op p  |  (ps)  |  [ps]  |  K { fps }  |  _  |  x@p
alt   ::=  p rhs  |  p rhs where ds
stmt  ::=  p <- e  |  e  |  let ds
f     ::=  x = e
fp    ::=  x = p
op    ::=  x  |  K
```

Our core language

prog	::=	**module** K **where** ds
d	::=	x = e
e	::=	x | K es | lit | e e | \x -> e | **let** ds **in** e | **case** e **of** alts
alt	::=	p -> e
p	::=	lit | K xs

Simple transformations

- Translating lists:

  translate [e_1, ..., e_n] = e_1 : ... : e_n

- Translating enumerations:

  translate [e_1 .. e_2] = enumFromTo e_1 e_n

- Translating infix applications:

  translate (e_1 op e_n) = op e_1 e_n

- Translating if-expressions:

  translate (**if** e_1 **then** e_2 **else** e_3) =

  case e_1 **of** True -> e_2; False -> e_3

List comprehensions

```
translate [ e | ]  =  e

translate [ e | e', stmts ]  =
      if e' then translate [ e | stmts ] else []

translate [ e | let ds, stmts ]  =
      let ds in translate [ e | stmts ]

translate [ e | p <- e', stmts ]  =
      let x p = translate [ e | stmts ]
          x _ = []
      in concat (map x e')      where x is a new variable
```

Pattern-matching

translate (**case** e **of** p_1 -> e_1 , ... , p_n -> e_n) =

let x = e **in** match [x] [\\$p_1$ -> e_1 , ... , \\$p_n$ -> e_n] (error "pmc")

where x is a new variable

translateDecl (f ps_1 = e_1 , ... , f ps_n = e_n) =

f = \xs -> match xs [\\$ps_1$ -> e_1 , ... , \\$ps_n$ -> e_n] (error "pmc")

where xs are new variables (of same length as each ps_i)

Function "match"

match xs ($funs_1$ ++ ... ++ $funs_n$) e_0

= match xs $funs_1$ (... (match xs $funs_n$ e_0) ...)

the mix rule

match (x:xs) [$\backslash y_1\ ps_1 \to e_1$, ... , $\backslash y_n\ ps_n \to e_n$] e_0

= match xs [$\backslash ps_1 \to [x/y_1]e_1$, ... , $\backslash ps_n \to [x/y_n]e_n$] e_0

the var rule

match [] [$\backslash \to e_1$, ... , $\backslash \to e_n$] e_0

= e_1 || ... || e_n || e_0

the null rule

Function "match"

the con rule

```
match (x:xs) (funs1 ++ ... ++ funsn) e0

   =  (case x of
          K1 ys1 -> match (ys1 ++ xs) (decon K1 funs1) fail
              ...
          Kn ysn -> match (ysn ++ xs) (decon Kn funsn) fail)  ||  e0

   where ys1 ... ysn are new variable lists of correct length

decon K [ \(K qs1) : ps1 -> e1 , ... , \(K qsm) : psm -> em ]

   =  [ \qs1++ps1 -> e1 , ... , \qsm++psm -> em ]
```

fail and fatbar ($\|$)

New abstract syntax forms introduced during translation of of pattern-matching.

Semantics:

$$\textbf{fail} \;\|\; e = e$$

$$e \;\|\; \textbf{fail} = e$$

$$e_1 \;\|\; e_2 = e_1 \qquad \text{if } e_1 \text{ cannot evaluate to } \textbf{fail}$$

$$e_1 \;\|\; e_2 = [e_2/\textbf{fail}]e_1 \qquad \text{(if functions can't return } \textbf{fail}\text{)}$$

Pattern-match example

```
zip [] bs            = []
zip (a:as) []        = []
zip (a:as) (b:bs)    = (a,b) : zip as bs
```

Pattern-match example

```
zip = \x1 x2 -> match [x1,x2]
                      [ \[] bs -> [],
                        \(a:as) [] -> [],
                        \(a:as) (b:bs) -> (a,b) : zip as bs ]
                      (error "pmc")
```

con rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> match [x2] [ \bs -> [] ] fail
    x3:x4 -> match [x3,x4,x2]
                    [ \a as [] -> [],
                      \a as (b:bs) -> (a,b) : zip as bs ]
                    fail
                  ) || (error "pmc")
```

var rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> match [] [ \ -> [] ] fail
    x3:x4 -> match [x2]
                   [ \[] -> [],
                     \(b:bs) -> (x3,b) : zip x4 bs ]
                   fail
                 ) || (error "pmc")
```

null rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []  ||  fail
    x3:x4 -> match [x2]
                   [ \[] -> [],
                     \(b:bs) -> (x3,b) : zip x4 bs ]
                    fail
                 ) || (error "pmc")
```

con rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []  ||  fail
    x3:x4 -> (case x2 of
                 [] -> match [] [ \ -> [] ] fail
                 x5:x6 -> match [x5,x6]
                                [ \b bs -> (x3,b) : zip x4 bs ]
                                fail
                 )  ||  fail
                    ) || (error "pmc")
```

null rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []  ||  fail
    x3:x4 -> (case x2 of
                 [] -> []  ||  fail
                 x5:x6 -> match [x5,x6]
                              [ \b bs -> (x3,b) : zip x4 bs ]
                              fail
                 )  ||  fail
                   ) || (error "pmc")
```

var rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []  ||  fail
    x3:x4 -> (case x2 of
                [] -> []  ||  fail
                x5:x6 -> match []
                             [ \ -> (x3,x5) : zip x4 x6 ]
                             fail
                )  ||  fail
                  ) || (error "pmc")
```

null rule applies

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []  ||  fail
    x3:x4 -> (case x2 of
                [] -> []  ||  fail
                x5:x6 -> (x3,x5) : zip x4 x6  ||  fail
               )  ||  fail
                 ) || (error "pmc")
```

semantics of ||

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []
    x3:x4 -> (case x2 of
                 [] -> []
                 x5:x6 -> (x3,x5) : zip x4 x6
                ) || fail
                   ) || (error "pmc")
```

semantics of ||

Pattern-match example

```
zip = \x1 x2 -> (case x1 of
    [] -> []
    x3:x4 -> case x2 of
               [] -> []
               x5:x6 -> (x3,x5) : zip x4 x6
                 ) || (error "pmc")
```

semantics of ||

Pattern-match example

```
zip = \x1 x2 -> case x1 of
    [] -> []
    x3:x4 -> case x2 of
                 [] -> []
                 x5:x6 -> (x3,x5) : zip x4 x6
```

Pattern-match example

```
zip [] bs             =  []
zip as []             =  []
zip (a:as) (b:bs)     =  (a,b) : zip as bs
```

Pattern-match example

```
zip = \x1 x2 -> match [x1,x2]
                      [ \[] bs -> [],
                        \as [] -> [],
                        \(a:as) (b:bs) -> (a,b) : zip as bs ]
                      (error "pmc")
```

First patterns are neither all <u>var</u> nor all <u>con</u> — use the mix rule!

Pattern-match example

```
zip = \x1 x2 ->
    match [x1,x2]
          [ \[] bs -> [] ]
          (match [x1,x2]
                 [ \as [] -> [] ]
                 (match [x1,x2]
                        [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                        (error "pmc")))
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> match [x2] [ \bs -> [] ] fail
        x3:x4 -> match [x2] [] fail
    ) || match [x1,x2]
                [ \as [] -> [] ]
                (match [x1,x2]
                    [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                    (error "pmc"))
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> match [] [ \ -> [] ] fail
        x3:x4 -> match [] [] fail
    ) ‖  match [x1,x2]
              [ \as [] -> [] ]
              (match [x1,x2]
                    [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                    (error "pmc"))
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> [] || fail
        x3:x4 -> fail
    ) ||  match [x1,x2]
                [ \as [] -> [] ]
                (match [x1,x2]
                      [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                      (error "pmc"))
```

Pattern-match example

```
zip = \x1 x2 ->
  (case x1 of
     [] -> [] || fail
     x3:x4 -> fail
  ) || match [x2]
             [ \[] -> [] ]
             (match [x1,x2]
                    [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                    (error "pmc"))
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> [] ‖ fail
        x3:x4 -> fail
    ) ‖ (case x2 of
            [] -> match [] [ \ -> [] ] fail
            x5:x6 -> match [] [] fail
        ) ‖ match [x1,x2]
                [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> [] || fail
        x3:x4 -> fail
    ) || (case x2 of
            [] -> [] || fail
            x5:x6 -> fail
         ) || match [x1,x2]
                    [ \(a:as) (b:bs) -> (a,b) : zip as bs ]
                   (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> [] ‖ fail
        x3:x4 -> fail
    ) ‖ (case x2 of
            [] -> [] ‖ fail
            x5:x6 -> fail
        ) ‖ (case x1 of
                [] -> match [x2] [ ] fail
                x7:x8 -> match [x7,x8,x2]
                            [ \a as (b:bs) -> (a,b) : zip as bs ]
                            fail
            ) ‖ (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
  (case x1 of
     [] -> [] ‖ fail
     x3:x4 -> fail
  ) ‖ (case x2 of
         [] -> [] ‖ fail
         x5:x6 -> fail
      ) ‖ (case x1 of
             [] -> fail
             x7:x8 -> match [x2]
                                 [ \(b:bs) -> (x7,b) : zip x8 bs ]
                                 fail
          ) ‖ (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> [] || fail
        x3:x4 -> fail
    ) || (case x2 of
            [] -> [] || fail
            x5:x6 -> fail
        ) || (case x1 of
                [] -> fail
                x7:x8 -> case x2 of
                            [] -> fail
                            x9:x10 -> (x7,x9) : zip x8 x10 || fail
            ) || (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    (case x1 of
        [] -> []
        x3:x4 -> fail
    ) || (case x2 of
            [] -> []
            x5:x6 -> fail
         ) || (case x1 of
                 [] -> fail
                 x7:x8 -> case x2 of
                            [] -> fail
                            x9:x10 -> (x7,x9) : zip x8 x10
              ) || (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    case x1 of
        [] -> []
        x3:x4 -> case x2 of
            [] -> []
            x5:x6 -> (case x1 of
                [] -> fail
                x7:x8 -> case x2 of
                            [] -> fail
                            x9:x10 -> (x7,x9) : zip x8 x10
                    ) || (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    case x1 of
        [] -> []
        x3:x4 -> case x2 of
            [] -> []
            x5:x6 -> (case x2 of
                        [] -> fail
                        x9:x10 -> (x3,x9) : zip x4 x10
                    ) || (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    case x1 of
        [] -> []
        x3:x4 -> case x2 of
            [] -> []
            x5:x6 -> ((x3,x5) : zip x4 x6) ‖ (error "pmc")
```

Pattern-match example

```
zip = \x1 x2 ->
    case x1 of
        [] -> []
        x3:x4 -> case x2 of
            [] -> []
            x5:x6 -> (x3,x5) : zip x4 x6
```

Summary

- Goal: transform rich abstract syntax trees into a simpler but equivalent syntactic subset
- Means: local rewrite rules of varying difficulty
- Challenge 1: define rules for full input syntax (see the Haskell Report ch. 3 for inspiration!)
- Challenge 2: apply rules to every subtree
- Challenge 3: organize into one or more passes