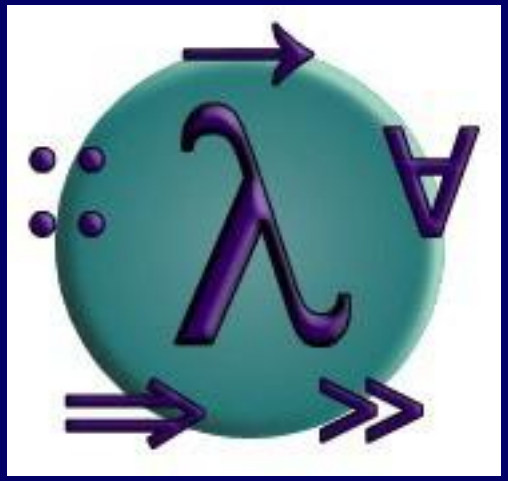
PROGRAMMING IN HASKELL
プログラミングHaskell

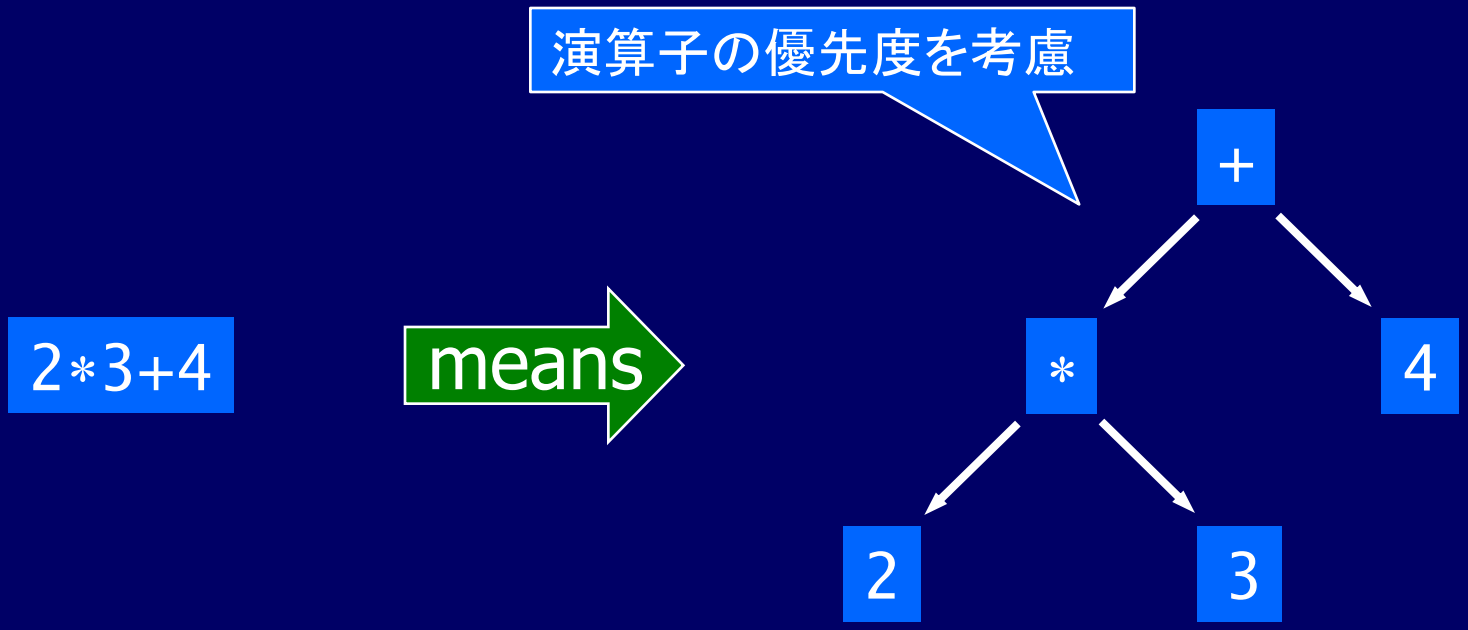


Chapter 8 - Functional Parsers
関数型パーサー

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

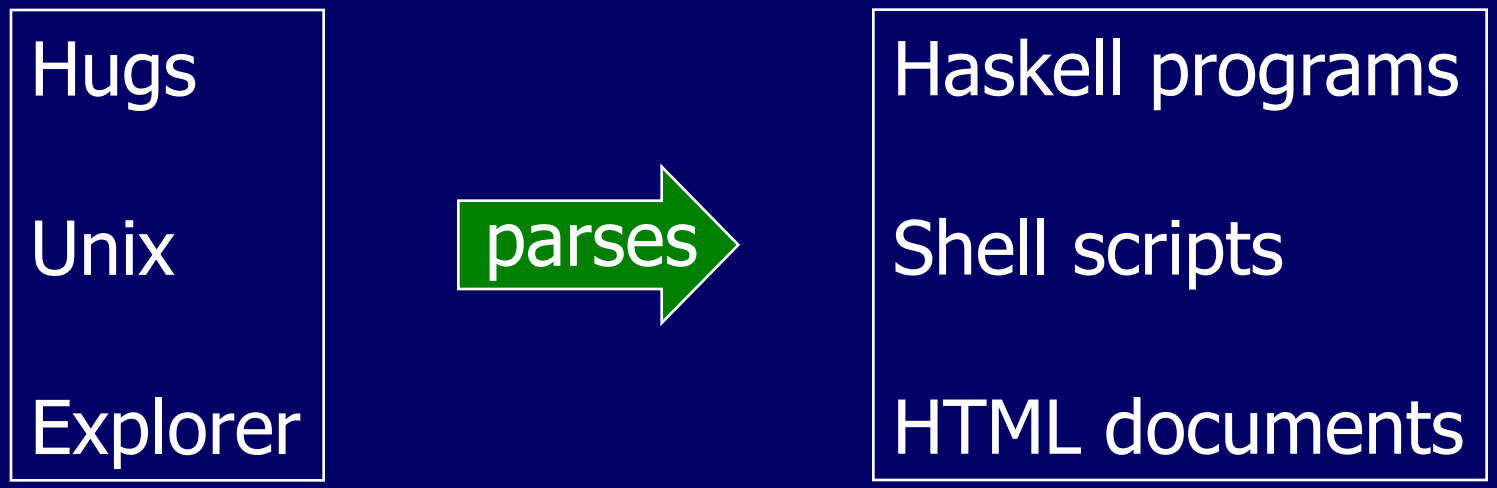
パーサーとは何か?

パーサー(parser, 構文解析器)は、文字列を受け取り、それを解析して、文法的な構造を表す構文木を返す



どこで使われているのか?

ほとんどの実用的なプログラムは、入力を前処理するためのパーサーを備えている



パーサーの型

Haskell のような関数型言語では、
パーサーを以下の型を持つ関数とみなすのが自然

型 Tree の定義は利用者が与える

```
type Parser = String → Tree
```

パーサーは、文字列を引数に取り、
何らかの形式の木構造を返す関数

パーサーは入力文字列の全てを必要とするとは限らないので、読まなかった文字列も返すことに:

どこまで読んだか分かる

```
type Parser = String → (Tree,String)
```

1 つの文字列に複数の解釈(解釈不可能も含む)が可能なので、返り値を結果のリストに一般化する:

```
type Parser = String → [(Tree,String)]
```

最後に、パーサーは木を返すとは限らないので、
任意の型に一般化する:

```
type Parser a = String → [(a,String)]
```

Parser a は型 a を返すパーサーの型
a は木構造のことが多いが、文字、
文字列、数値などの場合もある

Note:

- この章ではパーサーの戻り値を単純化する
  - 失敗:　　空リスト
  - 成功(受理): 要素が 1 つのリスト(singleton list)

基本的なパーサー

- パーサー item は入力が空文字列のとき失敗し、それ以外は、先頭 1 文字を消費し結果として返す:

文字を返すパーサー

```
item :: Parser Char
item  = λinp → case inp of
                 []     → []
                 (x:xs) → [(x,xs)]
```

- パーサー failure は常に失敗する:

入力を消費しない

```
failure :: Parser a
failure  = λinp → []
```

- パーサー　return v は、常に成功して、
  入力を消費することなく値 v を返す:

```
return  :: a → Parser a
return v = λinp → [(v,inp)]
```

- パーサー p +++ q は、p が成功した場合は p として、失敗した場合は q として振舞う(選択):

```
(+++)  :: Parser a → Parser a → Parser a
p +++ q = λinp → case p inp of
                   []         → parse q inp
                   [(v,out)] → [(v,out)]
```

既存パーサーから、新しいパーサーを構成する

- 関数 parse は、パーサー p を文字列に適用する:

```
parse :: Parser a → String → [(a,String)]
parse p inp = p inp
```

パーサーではなく、パーサーのドライバー

パーサーの動作例

- 5 つのパーサー部品の動作を簡単な例で示す:
  - 基本のパーサー部品: item, failure, return
  - パーサーコンビネータ(選択): +++
  - ドライバー:

既存パーサーから、新しいパーサーを構成する

```
parse :: Parser a → String → [(a,String)]
```

```
% hugs Parsing

> parse item ""
[]

> parse item "abc"
[('a',"bc")]
```

```
> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

Note:

- この例を実行するのに必要なライブラリ Parsing は、原著 "Programming in Haskell" の Web ページから入手できる
- 技術的な理由から、failure の最初の例は、実際には型エラーとなる。しかし、これほど単純でない通常の使用ではこの問題は起きない。
- Parser 型はモナドであり、多くの異なった種類の計算をモデル化するのに有用であると実証された数学的な構造

モナドのことは、教科書 p.136 まで気にしないこととする

パーサーの列(連結)

既存パーサーから、新しいパーサーを構成する

ひとつながりのパーサー群は、予約語 do を用いて一つの合成パーサーに結合することができる

例:

```
p :: Parser (Char,Char)
p  = do x ← item
        item
        y ← item
        return (x,y)
```

2 文字目を読み捨てて、1 文字目と 3 文字目をタプルにして返すパーサー

Note:

- レイアウト規則が適用されるように、個々のパーサーを厳密に同じインデントにすること
- 途中のパーサーが返す値は捨てられるが、値が必要な場合は、演算子 ← を用いて名前を付ける
- 最後のパーサーが返す値が、パーサー列全体の返す値となる

- パーサー列中のいずれかが失敗すると、列全体が失敗する

  例:

```
> parse p "abcdef"
[(('a','c'),"def")]

> parse p "ab"
[]
```

- do 記法は Parser 型に特有ではなく、任意のモナド型で使える

do の一般形

既存パーサーから、新しいパーサーを構成する

- 入力文字列を、n 回解析して、
  結果を変数 v1～vn に記録して、
  最後に関数 f をそれらに適用した結果を返す
- 値を参照しないとき、“vn ←” は不要
- pi が消費しなかった入力は、自動的に p(i+1) へ
- 途中で解析が失敗すると、自動的に全体の解析も失敗へ

```
do v1 ← p1
   v2 ← p2
   …
   vn ← pn
   return (f v1 v2 … vn)
```

```
do { v1 ← p1;
     v2 ← p2;
     …
     vn ← pn;
     return (f v1 v2 … vn) }
```

導出されたパーサーの部品

- 与えられた述語を満たす 1 文字を受理する:

```
sat  :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then
              return x
            else
              failure
```

- 数字 1 文字(digit)、指定された 1 文字(char)を受理する:

```
digit :: Parser Char
digit  = sat isDigit
char  :: Char → Parser Char
char x = sat (x ==)
```

- 指定された文字列を受理する:

```
string        :: String → Parser String
string []      = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

既存パーサーから、新しいパーサーを構成する
many と many1 は相互再帰している

- 与えられたパーサーを 0 回以上適用する:

```
many  :: Parser a → Parser [a]
many p = many1 p +++ return []
```

- 与えられたパーサーを 1 回以上適用する:

```
many1  :: Parser a → Parser [a]
many1 p = do v  ← p
             vs ← many p
             return (v:vs)
```

Example

"[1,2,3]" のような文字列を受理する

1 つ以上の数字のリストを受理し、それらを連結して
文字列として返すパーサー:

```
p :: Parser String
p  = do char '['
        d  ← digit
        ds ← many (do char ','
                      digit)
        char ']'
        return (d:ds)
```

For example:

```
> parse p "[1,2,3,4]"
[("1234","")]

> parse p "[1,2,3,4"
[]
```

Note:

- より洗練されたパーサーライブラリは、入力文字列の構文誤りを指摘し、回復することができる

数式

- 簡単な数式を考えてみよう

- 数は数字 1 文字
- 演算子として、加算 +、乗算 *、括弧 ( )

- 以下を仮定する
  - + と * は右結合
    - 右結合: 1 + 2 + 3 = 1 + (2 + 3)
    - 左結合: 1 + 2 + 3 = (1 + 2) + 3
  - * は + より優先度が高い

普通は左結合だけど

値を求めると同じだが、
式の形は異なる(演算子が
減算だと値も異なることに)

この数式の構文は、以下の素朴な文脈自由文法
によって形式的に定義される(最初の構文規則 p.99):

```
expr   → expr '+' expr
       | expr '*' expr
       | '(' expr ')'
       | digit

digit  → '0' | '1' | … | '9'
```

教科書は digit ではなく nat

* は + よりも優先度が高いことを考慮していない
2 * 3 + 4 を 2 * (3 + 4) と解析することも可能

(文脈自由)文法(補足)

2 年生は並行開講する講義で形式言語論を習う(がーん、なんてこった、ひょっとしてこっちが先?)

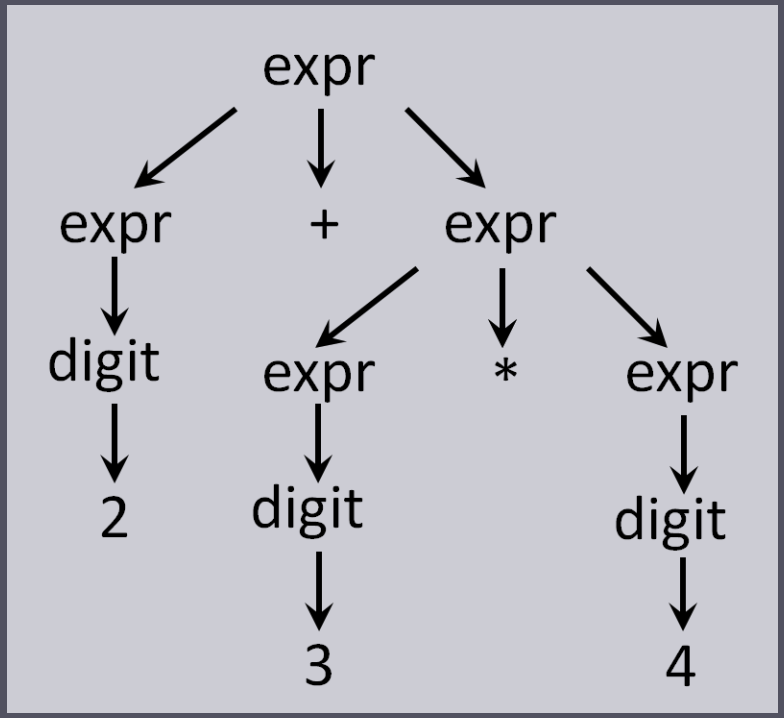
- 正しい文か否かの判定

```
expr → expr + expr
     → digit + expr
     → 2 + expr
     → 2 + expr * expr
     → 2 + digit * expr
     → 2 + 3 * expr
     → 2 + 3 * digit
     → 2 + 3 * 4
```

  - expr から 2 + 3 * 4 を導出できるので正しい文
  - 2 + + 3 は導出できないので正しい文ではない

(文脈自由)文法

- 実際には、文に対して構文木の構築を試みる
- 構文木ができれば、正しい文である
- 構文木ができれば、導出過程も分かる
- 2 + 3 * 4 に対する構文木



優先度を考慮した文脈自由文法 (2 番目の構文規則 p.100 上):

```
expr:   加法のレベル
term:   乗法のレベル
factor: 数字と括弧式のレベル
```

```
expr   → expr '+' expr | term

term   → term '*' term | factor

factor → digit | '(' expr ')'

digit  → '0' | '1' | … | '9'
```

演算子が右結合であることを考慮していない
2 + 3 + 4 を (2 + 3) + 4 と解析することも可能

2 番目の構文規則(補足)

- expr = expr + expr
  = expr + expr + expr
  = term + term + term
  - この時点の末端は term を + で繋いだ式
  - term は * を持つ式になりうる
  - そのとき、* は + よりも高い優先度を持つ(構文木で内側になる)
- expr = term
  = term * term
  = term * term * term
  = factor * factor * factor
  - この時点の末端は factor を * で繋いだ式
  - factor から expr を導出できないから、+ の式にはならない(括弧内を除く)
  - そのため、+ は * よりも高い優先度を持たない

expr は * とは並ばない

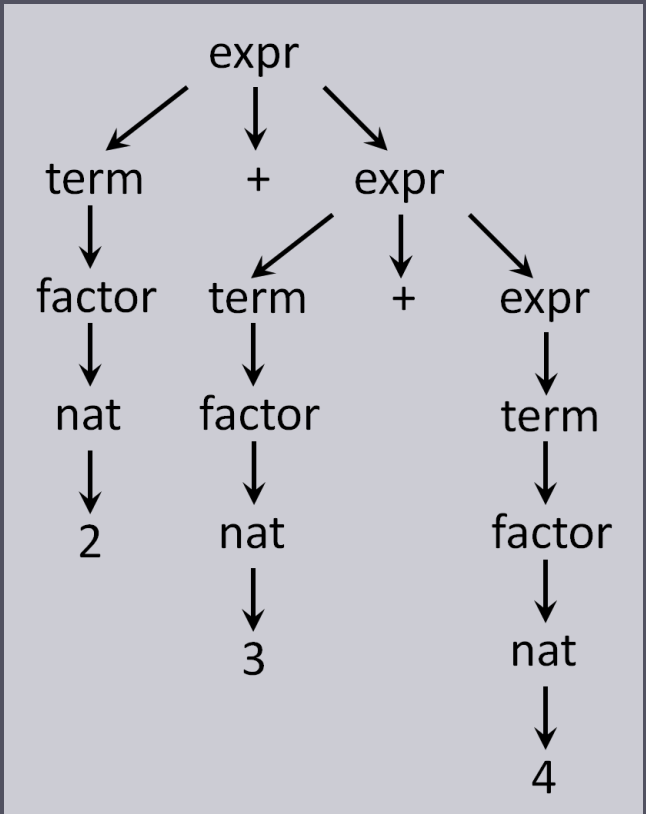
優先度と右結合性を考慮した文脈自由文法 (3 番目の構文規則 p.100 下、完成版):

```
expr   → term '+' expr | term

term   → factor '*' term | factor

factor → digit | '(' expr ')'

digit  → '0' | '1' | … | '9'
```

expr と term の自己再帰を第 2 引数だけで行う
自己再帰した部分は、構文木の内側になり、強く結合するため右結合となる

構文木(3 番目の構文規則)

- 2 + 3 + 4 の構文木
  - 2 + (3 + 4) として解析されている(右結合)
  - (2 + 3) + 4 ではないことに注意



効率化のために、expr と term に関する規則を factorise することが重要:

前:

```
expr   → term '+' expr | term
term   → factor '*' term | factor
```

後:

```
expr → term ('+' expr | ε)
term → factor ('*' term | ε)
```

共通項の括り出し

注意:

- 記号 ε は空文字列を表す

文法規則をパーサー部品を使って表すように書き換えて、
この文法を「式を評価する」パーサーに変換する

結果を以下に示す:

```
expr → term ('+' expr | ε)
expr :: Parser Int
expr  = do t ← term
           do char '+'
              e ← expr
              return (t + e)
            +++ return t
```

```
term → factor ('*' term | ε)
term :: Parser Int
term  = do f ← factor
           do char '*'
              t ← term
              return (f * t)
            +++ return f
```

```
factor → digit | '(' expr ')'
factor :: Parser Int
factor  = do d ← digit
             return (digitToInt d)
           +++ do char '('
                  e ← expr
                  char ')'
                  return e
```

最後に、以下のように定義すると、

```
eval    :: String → Int
eval xs = fst (head (parse expr xs))
```

例を試すことができる:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14
```

まとめ(8章)

- パーサー: 文字列を解析して構造(構文木)を返す関数
  - `type Parser a = String → [(a,String)]`
  - Parser a は a 型を返すパーサーの型
    - 失敗: 空リスト
    - 成功(受理): 要素が 1 つのリスト
- パーサーの部品
  - item (1 文字), failure (失敗), return v (v を返す)
  - sat p (述語 p を満たす1文字), digit (数字 1 文字), char x (指定された 1 文字), string str (指定された文字列)
  - パーサーコンビネータ: 部品から新しいパーサーを構成する
    - +++ (選択), many (0 回以上の反復), many1 (1 回以上の反復), do { x ← p1; p2; ...; y ← pn; return (x, y) } (連結)
  - ドライバー `parse :: Parser a → String → [(a,String)]`

動作確認方法

- http://www.cs.nott.ac.uk/~gmh/book.html の Code から以下のファイルをダウンロードする
  - Parsing.lhs (Functional parsing library, 8.7 節まで)
  - parser.lhs (Expression parser, 8.8 節)
- Parsing.lhs と parser.lhs を同じディレクトリに置く
  - parser.lhs が Parsing.lhs を import している
- ghci で parser.lhs をロードすれば、教科書の例の動作を確認できる
  - ただし、p.93 のパーサー p (文字を 3 つ解析) と p.98 のパーサー p (自然数のリストを解析) は含まれない

ここから >>= の説明が始まるが、
教科書 p.136 まで後回しにすること

モナド型(教科書p.136)

- return と (>>=) を備えた型は、クラス Monad のインスタンスである(になれる)

```
class Monad m where
  return  :: a → m a
  (>>=)   :: m a → (a → m b) → m b
```

  - (==) と (/=) を備えた型は、クラス Eq のインスタンスである(になれる)のと同じ
  - 型 m は型変数 a を取る(m a や m b に注意)
  - Parser (8章)も IO (9章)も実はモナドのインスタンス

```
instance Monad Parser where …

instance Monad IO wehre …
```

Parser の例による (>>=) の説明

- (>>=) は、Parser a と (a → Parser b) から Parser b を生成

```
(>>=) :: Parser a → (a → Parser b) → Parser b
p >>= f   = ¥inp → case parse p inp of
                      [] → []
                      [(v,out)] -> parse (f v) out
```

- パーサー p >>= f は、
  - 入力文字列 inp を p で解析 parse p inp し、
  - 失敗 [] の場合、全体も失敗 []、
  - それ以外の場合(結果を (v,out) とする)、
    - f を v に適用して新しいパーサー f v を生成し、
    - out をそのパーサー f v で解析 parse (f v) out し、
    - 全体の結果とする

ポイント1: 逐次実行
(1) p で inp を解析し、
(2) 残りの入力 out を f v で解析し、
(3) 全体の結果とする

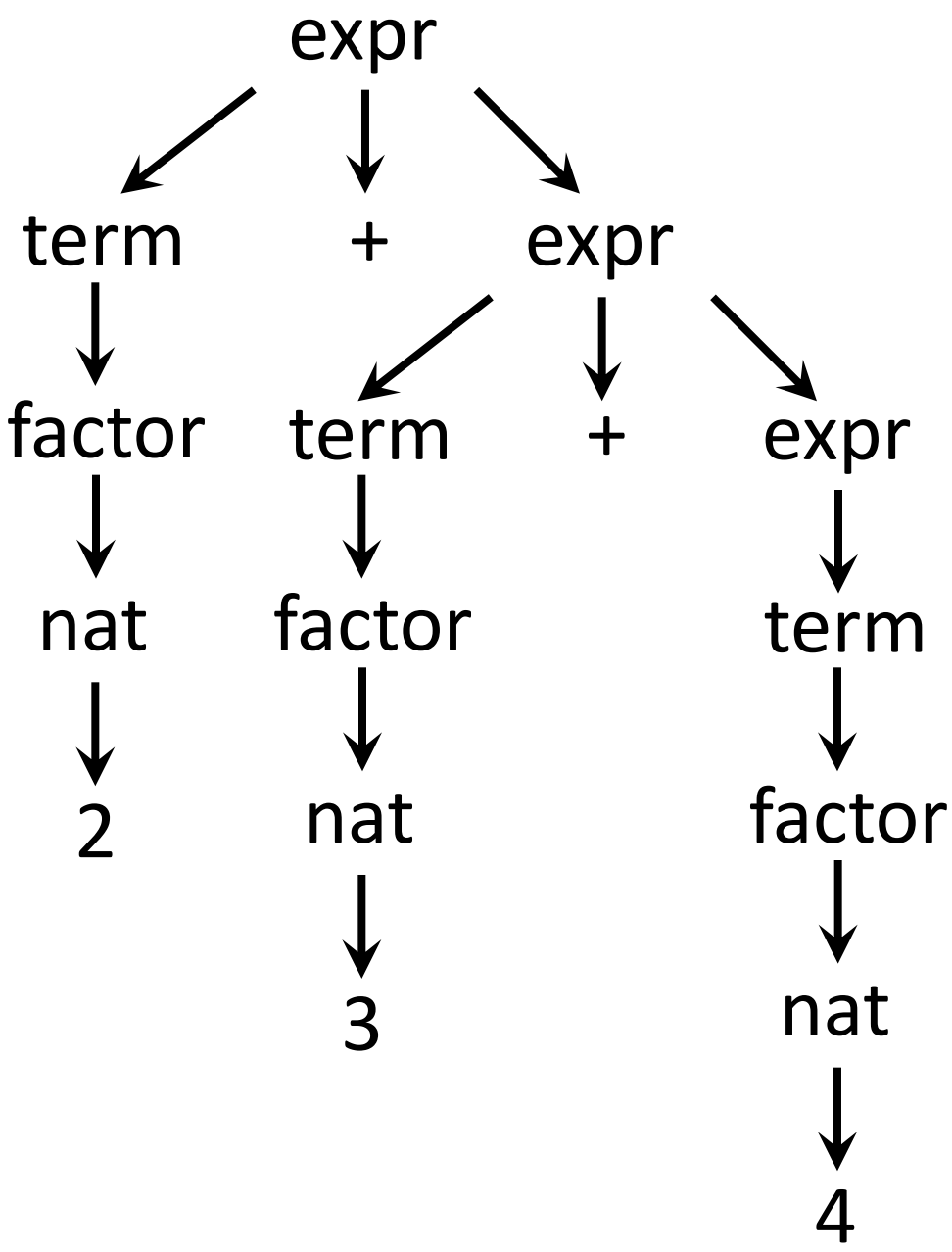
ポイント2: エラー処理
(4) 個別にエラー処理をしなくても良い

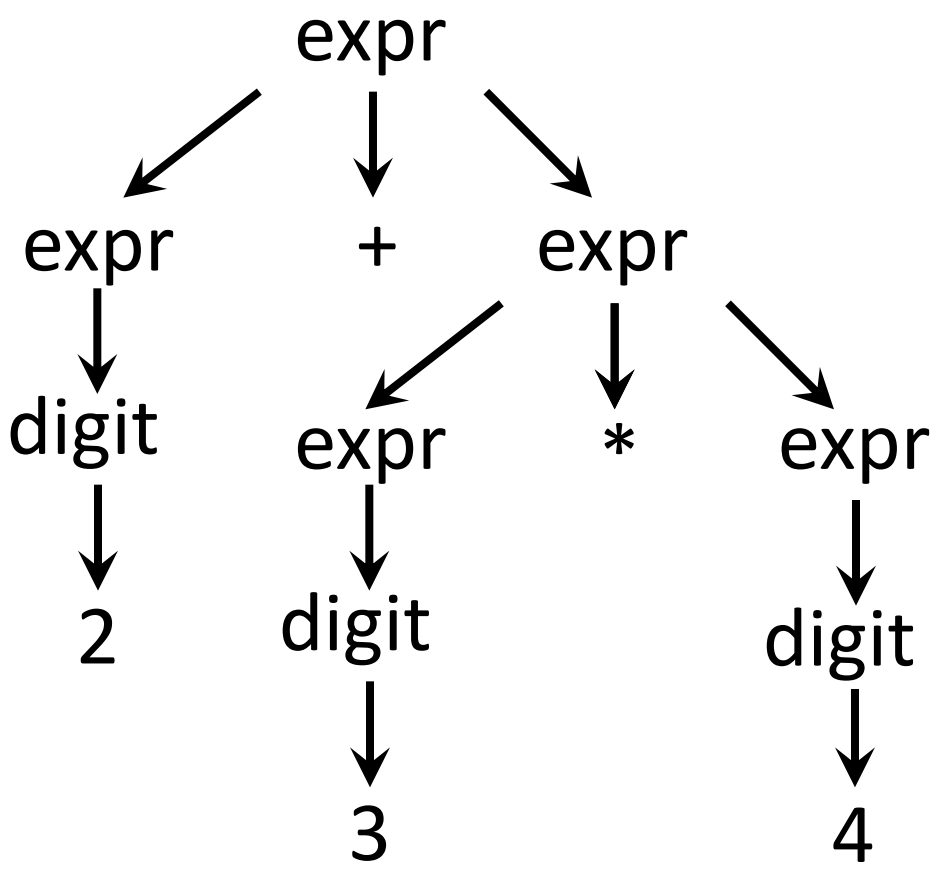
parse (p1 >>= λv1 → ⋯)

- parse (p1 >>= λv1 → retrun v1) text の動作
  - p1 で text を解析し、結果を (a1,out1) とする
  - λv1 → retrun v1 を a1 に適用して、新しいパーサー return a1 を生成する
  - 新しいパーサーで out1 を解析する
    - return a1 で out1 を解析する
    - その結果、 [(a1, out1)] が返る
    - 注意: return は入力を消費しない

parse (p1 >>= λv1 → (p2 >>= λv2 → ･･･))

- parse (p1 >>= λv1 → (p2 >>= λv2 → return (v1,v2))) text の動作
  - p1 で text を解析し、結果を (a1,out1) とする
  - λv1 → (p2 >>= λv2 → return (v1,v2)) を a1 に適用して新しいパーサー (p2 >>= λv2 → return (a1,v2)) を生成する
  - 新しいパーサーで out1 を解析する
    - p2 で out1 を解析し、結果を (a2,out2) とする
    - λv2 → return (a1,v2) を a2 に適用して新しいパーサー return (a1,a2)) を生成する
    - 新しいパーサーで out2 を解析する
      - return (a1,a2)) で out2 を解析する
      - [((a1,a2), out2)] が返る


expr
term
+
expr
factor
term
+
expr
nat
factor
term
2
nat
factor
3
nat
4


expr
expr
+
expr
digit
expr
*
expr
2
digit
digit
3
4