1. コア言語

1. コア言語の非形式的導入
2. コア言語の抽象構文
3. コア言語でのプログラムおよび式に対応するHaskellのデータ型 `CoreProgram` および `CoreExpr` .
4. コア言語の関数をまとめた**標準プレリュード**
5. プリティプリンタ
6. パーザ（構文解析器）

1.1 コア言語の概要

```
main = double 21 ;
double x = x + x
```

- コアプログラムは，スーパーコンビネータ定義の集まり
- コアプログラムの実行は，`main` を評価する
- 関数はスーパーコンビネータで定義する
- スーパーコンビネータにはCAFも含まれる

1.1.1 局所定義

- `let`（非再帰）または `letrec`（再帰）を使う
- `let` と `letrec` を区別するのは，`let` は `letrec` より実装が単純にできる
- バインディングの左辺は単一の変数

1.1.2 λ抽象

コンパイルの前処理としてλ持ち上げをおこなって、トップレベルでスーパーコンビネータとして定義する

1.1.3 構造をもつデータ型

代数データ型：

Haskell:

```
data Colour = Red ¦ Green ¦ Blue
data Complex = Rect Double Double ¦ Polar Double Double
data Tree a = Leaf a ¦ Branch (Tree a) (Tree a)
```

アプローチ方法:

- データコンストラクタを単純かつ統一的に表現
- パターン照合を単純な `case` 式に変換する

1.1.4 コンストラクタの表現

tag をコンストラクタ識別子（整数）、$arity$ をそのコンストラクタのアリティとして、
`Pack{`$tag, arity$`}` のように表現する

```
Red    = Pack{1,0}
Green  = Pack{2,0}
Blue   = Pack{3,0}

Rect   = Pack{4,2}
Polar  = Pack{5,2}

Leaf   = Pack{6,1}
Branch = Pack{7,2}
```

データ型を、実行時に区別する必要はないので、

```
Red    = Pack{1,0}
Green  = Pack{2,0}
Blue   = Pack{3,0}

Rect   = Pack{1,2}
Polar  = Pack{2,2}

Leaf   = Pack{1,1}
Branch = Pack{2,2}
```

でよい

`case` 式

```
isRed c = case c of
              <1> -> True  ;
              <2> -> False ;
              <3> -> False
```

とか、

```
depth t = case t of
              <1> n     -> 0
              <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

入れ子のパターンはサポートしない

1.2 コア言語の構文

優先順位	結合方向	演算子
6	左	適用
5	右	*
	無	/
4	右	+
	無	-
3	無	== ~= > >= < <=
2	右	&
1	右	|

1.3 コア言語のデータ型

抽象構文木

```
module Language where
import Utils

data Expr a
  = EVar Name
  | ENum Int
  | EConstr Int Int
  | EAp (Expr a) (Expr a)
  | ELet
      IsRec
      [(a, Expr a)]
      (Expr a)
  | ECase
      (Expr a)
      [Alter a]
  | ELam [a] (Expr a)
  deriving (Show)
```

抽象構文木

- `Expr` はバインダー（変数出現を束縛時の名前）でパラメータ化する
- 束縛位置では `Name` を使う
- 通常使う型名は、以下のようにする

```
type CoreExpr = Expr Name
type Name = String
```

抽象構文木

- 変数は、名前で区別する（`EVar Name`）
- 数は、整数のみ（`ENum Int`）
- データコンストラクタ（データ構成子）は §1.1.4 での議論どおり、構成子IDとアリティで表現（`EConstr Int Int`）
- 関数適用では関数と引数を子として並べて表現する。
- 中置演算子適用も抽象構文木では関数適用と同じ
  - `x + y` は
    `EAp (EAp (EVar "+") (EVar "x")) (EVar "y")`
    と表現する

`let` 式

```
type IsRec = Bool
recursive, nonRecursive :: IsRec
recursive    = True
nonRecursive = False

bindersOf :: [(a, b)] -> [a]
bindersOf defns = [ name ¦ (name, rhs) <-defns ]

rhssOf    :: [(a, b)] -> [b]
rhssOf defns = [ rhs ¦ (name, rhs) <- defns ]
```

`case` 式

```
ECase
  (Expr a)  -- 分析対象式
  [Alter a] -- 選択肢リスト
```

選択肢は、タグ、束縛変数リスト、矢印右辺の（選択される）式の三つ組

```
type Alter a = (Int, [a], Expr a)
type CoreAlter = Alter Name
```

アトミックな式かを判定する述語

```
isAtomicExpr :: Expr a -> Bool
isAtomicExpr (EVar v) = True
isAtomicExpr (ENum n) = True
isAtomicExpr e        = False
```

プログラムはスーパーコンビネータ定義の集まり

```
type Program a = [ScDefn a]
type CoreProgram = Program Name
```

スーパーコンビネータは、名前と仮パラメータ名と本体の三つ組

```
type ScDefn a = (Name, [a], Expr a )
type CoreScDefn = ScDefn Name
```

```
main = double 21 ;
double x = x + x
```

というコアプログラムは、Haskellの以下のようなデータで表現される

```
[ ("main", [], EAp (EVar "double") (ENum 21))
, ("double", ["x"], EAp (EAp (EVar "+") (EVar "x")) (EVar "x"))
]
```

プログラム	$program$	$\rightarrow$	$sc_1\,\texttt{;}\,\ldots\,\texttt{;}\,sc_n$	$n \geq 1$
スーパーコンビネータ	sc	$\rightarrow$	$var\ var_1\ \ldots\ var_n\ \texttt{=}\ expr$	$n \geq 0$
式	$expr$	$\rightarrow$	$expr\ aexpr$	適用
		$\mid$	$expr_1\ binop\ expr_2$	中置二項演算子適用
		$\mid$	$\texttt{let}\ defns\ \texttt{in}\ expr$	局所定義
		$\mid$	$\texttt{letrec}\ defns\ \texttt{in}\ expr$	局所再帰定義
		$\mid$	$\texttt{case}\ expr\ \texttt{of}\ alt$	$case$式
		$\mid$	$\backslash\ var_1\ \ldots\ var_n\ \texttt{.}\ expr$	λ抽象 $n \geq 1$
		$\mid$	$aexpr$	アトミックな式
	$aexpr$	$\rightarrow$	var	変数
		$\mid$	num	数
		$\mid$	$\texttt{Pack\{}num\texttt{,}\ num\texttt{\}}$	コンストラクタ
		$\mid$	$\texttt{(}\ expr\ \texttt{)}$	括弧で囲まれた式

定義	$defns$	$\rightarrow$	$defn_1$; . . . ; $defn_n$	$n \geq 1$
	$defn$	$\rightarrow$	var = $expr$	
選択肢	$alts$	$\rightarrow$	alt_1 ; . . . ; alt_n	$n \geq 1$
	alt	$\rightarrow$	<num> var_1 . . . var_n -> $expr$	$n \geq 0$
二項演算子	$binop$	$\rightarrow$	$arithop$ | $relop$ | $boolop$	
	$arithop$	$\rightarrow$	+ | - | * | /	算術
	$relop$	$\rightarrow$	< | <>= | == | ~= | >= | >=	比較
	$boolop$	$\rightarrow$	& | |	論理
変数	var	$\rightarrow$	$alpha$ $varch_1$ . . . $varch_n$	$n \geq 0$
	$alpha$	$\rightarrow$	アルファベット文字	
	$varch$	$\rightarrow$	$alpha$ | $digit$ | _	
数	num	$\rightarrow$	$digit_1 \ldots digit_n$	$n \geq 1$

1.4 小さな標準プレリュード

```
I x = x ;
K x y = x ;
K1 x y = y ;
S f g x = f x (g x) ;
compose f g x = f (g x) ;
twice f = compose f f
```

```
preludeDefs :: CoreProgram
preludeDefs
  = [ ("I", ["x"], EVar "x")
    , ("K", ["x","y"], EVar "x")
    , ("K1",["x","y"], EVar "y")
    , ("S", ["f","g","x"], EAp (EAp (EVar "f") (EVar "x"))
                               (EAp (EVar "g") (EVar "x")))
    , ("compose", ["f","g","x"], EAp (EVar "f")
                                     (EAp (EVar "g") (EVar "x")))
    , ("twice", ["f"], EAp (EAp (EVar "compose") (EVar "f")) (EVar "f"))
    ]
```

1.5 コア言語プリティプリンタ

コア言語を`Show`クラスのインスタンスとしている（自動導出）ので、表示は可能だが、もうすこしなんとかしたい。 そこでプリティプリンタですよ。

```
pprint :: CoreProgram -> String
```

1.5.1 文字列を用いたプリティプリンタ

```
pprExpr :: CoreExpr -> String
pprExpr (ENum n)    = show n
pprExpr (EVar v)    = v
pprExpr (EAp e1 e2) = pprExpr e1 ++ " " ++ pprAExpr e2
```

`pprAExpr` は引数の式がアトミックではないときに括弧で囲う

```
pprAExpr :: CoreExpr -> String
pprAExpr e
  ¦ isAtomicExpr e = pprExpr e
  ¦ otherwise      = "(" ++ pprExpr e ++ ")"
```

`++` をがっつり使っているので、パフォーマンスがすぐだめになる。

`pprExpr` は最悪、式のサイズ n に対して $\Theta(n^2)$ の計算量になる。

抽象構文木で、左側が深い木を印字することを考えれば、理解できる。

```
{- ¦ 左側の深い構文木の生成 -}
mkMultiAp :: Int -> CoreExpr -> CoreExpr -> CoreExpr
mkMultiAp n e1 e2 = foldl EAp e1 (take n (repeat e2))
```

練習問題 1.1

さまざまな、n について、以下の式を評価するのに必要なステップ数を計測せよ

```
pprExpr (mkMulti n (EVar "f") (EVar "x"))
```

1.5.2 プリティプリント用抽象データ型

プリティプリントの問題を以下の2つに分解する

- 必要な操作は何か
- それらの操作の効率のよい実行方法は何か

抽象データ型（実装の詳細を抽象したデータ型）を考えることで ↑ を実現する ここでは抽象データ型 `Iseq` を考える。

操作

```
iNil :: Iseq                            -- ^ 空の Iseq
iStr :: String -> Iseq                  -- ^ 文字列から Iseq への変換
iAppend :: Iseq -> Iseq -> Iseq         -- ^ 2つの Iseq の連結
iNewline :: Iseq                        -- ^ 改行
iIndent  :: Iseq -> Iseq                -- ^ Iseq の字下げ
iDisplay :: Iseq -> String              -- ^ Iseq から文字列への変換
```

`pprExpr`

```
pprExpr :: CoreExpr -> Iseq
pprExpr (EVar v)    = iStr v
pprExpr (EAp e1 e2) = pprExpr e1 `iAppend` iStr " " `iAppend` pprAExpr e2

pprExpr (ELet isrec defns expr)
  = iConcat [ iStr keyword, iNewline
            , iStr "  ", iIndent (pprDefns defns), iNewline
            , iStr "in ", pprExpr expr
            ]
    where
      keyword ¦ not isrec = iStr "let"
              ¦ isrec     = iStr "letrec"

pprDefns :: [(Name, CoreExpr)] -> Iseq
pprDefns defns = iInterleave sep (map pprDefn defns)
  where
    sep = iConcat [ iStr ";", iNewline ]

pprDefn :: (Name, CoreExpr) -> Iseq
pprDefn (name, expr)
  = iConcat [ iStr name, iStr " = ", iIndent (pprExpr expr) ]
```

```
infixr 5 `iAppend`

iConcat      :: [Iseq] -> Iseq
iInterleave :: Iseq -> [Iseq] -> Iseq
```

練習問題 1.2

`iConcat` および `iInterleave` を `iAppend` と `iNil` を使って定義せよ

ほとんどのプリティプリンティング関数は、`Iseq`型の値を返し、`iDisplay`は最後にトップレベルで適用してプログラムを表示する。

```
pprint prog = iDisplay (pprProgram prog)
```

練習問題 1.3

- `pprExpr` が `case` 式とλ-抽象式を扱えるようにせよ
- `pprAExpr` および `pprProgram` を同様のスタイルで定義せよ

1.5.3 `Iseq` の実装

```
data Iseq = INil
          | IStr String
          | IAppend Iseq Iseq
```

データ構造を用いて操作の表現をするのは、最後に `iDisplay` が呼ばれるまで、仕事を先延ばしにするという意図がある。

```
iNil                = INil
iStr str            = IStr str
iAppend seq1 seq2 = IApend seq1 seq2
```

とりあえずインデントは無視する（次節で改良予定）ことにすると、`iIndent`と`iNewline`の定義は直截的に、

```
iIndent seq = seq
iNewline    = IStr "\n"
```

これで、`iDisplay` をどうするかがすべてということになる。`Iseq` をサイズに線形な計算量で文字列にできるようにするのが目標。`iDisplay` は `flatten` というより一般的な関数を使って定義する

```
flatten :: [Iseq] -> String

iDisplay seq = flatten [seq]
```

```
flatten []                = ""
flatten (INil : seqs)     = flatten seqs
flatten (Istr s : seqs) = s ++ flatten seqs
flatten (IAppend seq1 seq2 : seqs) = flatten (seq1 : seq2 : seqs)
```

練習問題 1.4

`Iseq` による `flatten` の計算量は、`Iseq` のサイズに対してどうなっているか。`pprExpr` を `Iseq` を返すようにしたうえで、練習問題 1.1 の実験を行って計測してみよ。 `pprExpr` の結果に `iDisplay` を適用することを忘れないように

練習問題 1.5

抽象データ型を採用するもう１つの利点は、抽象データ型の実装がインターフェイスに影響しないということである。 `iAppend` を再定義して、一方の引数が `INil` であったときに結果が単純になるようにせよ。

1.5.4 配置と字下げ

`iIndent` の実装を自明なものから、まともなものにしよう。 `Iseq` に `IIndent` と `INewline` を追加する。

```
data Iseq = INil
          | IStr String
          | IAppend Iseq Iseq
          | IIndent Iseq
          | INewline

iIndent :: Iseq -> Iseq
iIndent seq = IIndent seq

iNewline :: Iseq
iNewline = INewline
```

`flatten` をより強力なものにする。

- 現在のカラム位置を保持する
- `Iseq` とそのインデントレベルとの対のリストをワーキングリストとする

```
flatten :: Int                  -- ^ 現在のカラム； 0 は最初のカラム
        -> [(Iseq, Int)]        -- ^ ワークリスト
        -> String               -- ^ 結果
```

あわせて、`iDisplay` も変更

```
iDisplay :: Iseq -> String
iDisplay seq = flatten 0 [(seq, 0)]
```

```
flatten col ((INewline, indent) : seqs)
  = '\n' : space indent ++ flatten indent seqs
flatten col ((IIndent seq, indent) : seqs)
  = flatten col ((seq, col) : seqs)
```

練習問題 1.6

- `flatten` を `IAppend`、`IStr`、`INil` に対応させよ
- `pprExpr` を `ELet` を含む式に適用して、正しく配置されるかを確かめよ

練習問題 1.7

このプリティプリンタは、`IStr` が `'\n'` を含む文字列を持つとき正しく動作しない。
`iStr` を変更して、改行文字が `INewline` に置き換るようにせよ。

1.5.5 中置演算子の優先順位

中置演算子の適用は内部的には関数適用と同じなので、`pprExpr` で中置演算子の適用は中置記法に変換する必要がある。 そのためには、以下のように演算子ごとに用意する

```
pprExpr (EAp (EAp (EVar "+") e1) e2)
  = iConcat [ pprAExpr e1, iStr " + ", pprAExpr e2 ]
```

ただし、これでは、括弧が多すぎるので、二項演算子の優先順位を考慮した関数が必要になる。ひとつの方法は、そのコンテキストでの優先順位を示す引数を導入することである。

1.5.6 その他の `Iseq` 上の便利関数

```
iNum :: Int -> Iseq                -- ^ 数の表示
iNum n = iStr (show n)

iFWNum :: Int -> Int -> Iseq       -- ^ 固定幅に右寄せで数を表示
iFWNum width n
  = iStr (space (width - length digits) ++ digits)
    where
      digits = show n

iLayn :: [Iseq] -> Iseq            -- ^ リスト項目を番号付きで表示
iLayn seqs = iConcat (map lay_item (zip [1..] seqs))
  where
    lay_item (n, seq)
      = iConcat [ iFWNum 4 n, iStr ") ", iIndent seq, iNewline ]
```

1.5.7 まとめ

- 抽象データ型
  - データの構成を隠蔽、スマートコンストラクタをインターフェイスとする
  - Haskell ではモジュールを分離、データ構成子をエクスポートせず、スマートコンストラクタをエクスポートする
- 汎用化（generalisation）技法
  - `iDisplay` を `flatten` で表現するのがその例

1.6 コア言語の構文解析器

- コアプログラム（具象構文）を文字列として読む
- 文字列を字句解析器でトークン列にする

```
clex :: String -> [Token]
```

- トークン列を構文解析器でコアプログラム（抽象構文）として解読する

```
syntax :: [Token] -> CoreProgram
```

ここでは、

```
parse :: String -> CoreProgram
parse = syntax . clex
```

とする

1.6.1 字句解析

```
type Token = String          -- A token is never empty

clex (c:cs)
  ¦ isWhiteSpace c = clex cs
  ¦ isDigit c      = numToken : clex restCs
      where
        (numCs, restCs) = span isDigit cs
        numToken        = c : numCs
  ¦ isAlpha c      = varToken : clex restCs
      where
        (idCs, restCs) = span isIdChar cs
        varToken       = c : idCs
  ¦ otherwise      = [c] : clex cs
clex []            = []
```

補助関数

```
isWhiteSpace :: Char -> Bool
isIdChar     :: Char -> Bool

usWhiteSpace c = c `elem` " \t\n"
isIdChar c     = isAlpha c || isDigit c || c == '_'
```

練習問題 1.9

字句解析器を変更して、インラインコメント（`--` から行末までがコメント）を無視するようにせよ

練習問題 1.10

現在の字句解析器は2文字の中置演算子を認識できない コアプログラムで使う2文字演算子は `twoCharOps` であたえられる 字句解析器 clex を変更して、これらの演算子を認識できるようにせよ

```
twoCharOps :: [String]
twoChanOps = ["==", "~=", ">=", "<=", "->"]
```

練習問題 1.11

構文解析器がパーズエラー箇所の行番号を報告できるように、トークンに行番号を付加するように字句解析器を改造せよ

```
type Token = (Int, String)

clex :: Int -> String -> [Token]
```

1.6.2 構文解析のための基本ツール

`Parser` 型の定義

```
type Parser a = [Token] -> a   -- NG
```

1. 構文解析器を部品化するなら、1つの構文解析器が消費した残りのトークン列を返すべき
2. 文法に曖昧性があるときも、文法に沿った構文解析結果を返すためには構文解析結果をリストで表現すべき

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

小さな構文解析器の例

```
-- ¦
-- >>> pLit "hello" ["hello", "John", "!"]
-- [("hello", ["John","!"])]
pLit :: String -> Parser String
pLit s (tok : toks)
  ¦ s == tok  = [(s, toks)]
  ¦ otherwise = []
pLit _ []     = []
```

変数の構文解析器

```
pVar :: Parse String
pVar [] = []
pVar (tok:toks) = case tok of
  c:_ ¦ isAlpha c -> [(tok,toks)]
```

これではキーワードも変数としてしまう。 改良は練習問題 1.17

2つの構文解析器を選択肢とする

```
pAlt :: Parser a -> Parser a -> Parser a
pAlt p1 p2 toks = p1 toks ++ p2 toks
```

使用例

```
pHelloOrGoodbye :: Parser String
pHelloOrGoodbye = pLit "hello" `pAlt` pLit "goodbye"
```

2つの構文解析器を連続適用して結果を組み合わせる

```
pThen :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
pThen combine p1 p2 toks
  = [ (combine v1 v2, toks2) ¦ (v1, toks1) <- toks
                             , (v2, toks2) <- toks1 ]
```

単純な文法

$$
\begin{array}{lcl}
\mathit{greeting} & \rightarrow & \mathit{hg}\ \mathit{person}\ \texttt{!} \\
\mathit{hg} & \rightarrow & \texttt{hello} \\
 & | & \texttt{goodbye}
\end{array}
$$

構文解析器 `pGreeting`

```
-- ¦
-- >>> pGreeting ["goodbye", "James", "!"]
-- [(("goodbye","James"),["!"])]

pGreeting :: Parser (String, String)
pGreeting = pThen mkPair pHelloOrGoodbye pVar
  where
    mkPair hg name = (hg, name)
```

1.6.3 ツールを磨く

前頁の `pGreeting` は文法に準拠していない

改定版

```
pGreeting = pThen keepFirst
                 (pThen mkPair pHelloOrGoodbye pVar)
                 (pLit "!")
            where
              keepFirst = const
              mkPair    = (,)
```

以下のように書けるほうがわかりやすい

```
-- ¦
-- >>> pGreeting ["goodbye", "James", "!"]
-- [(("goodbye","James"),[])]

pGreeting = pThen3 mkGreeting
              pHelloOrGoodbye
              pVar
              (pLit "!")
            where
              mkGreeting hg name exlamation = (hg, name)
```

練習問題 1.12

- `pThen3` の型を与え、定義し、テストせよ
- `pThen4` （後で使う）を書け

繰り返し

```
pZeroOrMore :: Parser a -> Parser [a]
pZeroOrMore p = pOneOrMore p `pAlt` pEmpty []

pEmpty :: a -> Parser a
pEmpty = undefined

pOneOrMore :: Parser a -> Parser [a]
pOneOrMore = undefined
```

練習問題 1.13

- `pOneOrMore` および `pEmpty` の定義を書け（ヒント： `pOneOrMore` から `pZeroOrMore` を呼ぶとよい）

構文解析の結果を処理できるようにと嬉しい

```
pApply :: Paser a -> (a -> b) -> Parser b
```

練習問題 1.14

`pApply` を定義し、テストせよ

区切り子を挟んで並べられた記号を構文解析するという場面もよく現れる。たとえば、セミコロンで区切られたスーパーコンビネータ定義のならびがある。

```
pOneOrMoreWithSep :: Parser a -> Parser b -> Parser [a]
pOneOrMoreWithSep p sep = undefined
```

練習問題 1.15

`pOneOrMoreWithSep` を定義し、テストせよ

以下のような、文法を解析するのに役立つ

$$
\begin{array}{lcl}
program & \rightarrow & sc\ programRest \\
programRest & \rightarrow & ;\ program \\
 & | & \epsilon
\end{array}
$$

`pLit` および `pVar` の一般化

```
pSat :: (String -> Bool) -> Parser String
```

`pLit` は `pSat` を使って以下のように定義できる

```
pLit s = pSat (s ==)
```

練習問題 1.16

- `pSat` を定義し、テストせよ
- `pVar` を `pSat` を用いて定義せよ

練習問題 1.17

`pVar` の定義で、 `pSat` に渡す述語を変更して、キーワードを認識しないようにせよ

```
keywords :: [String]
keywords = ["let", "letrec", "in", "case", "of", "Pack"]
```

練習問題 1.18

数トークンを識別する

```
pNum :: Parser Int
```

を書け

練習問題 1.19

構文エラーをおこす `let` 式

```
f x = let x1 = x; x2 = x; ...; xn = x
      of x1
```

を $n = 5, 10, 15, 20$のようにnを増やしたときに簡約ステップ数はどのように増加するか。

```
pOneOrMore (pLit "x") ["x", "x", "x", "x", "x", "x"]
```

を評価してみよ。

余分な候補を除去するように `pOneOrMore` を定義せよ

1.6.4 コア言語の構文解析

`syntax` は `pProgram` の結果から `CoreProgram` をとりだす

```
syntax = takeFirstParse . pProgram
  where
    takeFirstParse ((prog, []) : others) = prog
    takeFirstParse (parse      : others) = takeFirstParse others
    takeFirstParse other                 = error "syntax error"
```

`pProgram`

```
pProgram :: Parser CoreProgram
pProgram = pOneOrMoreWithSep pSc (pLit ";")

pSc :: Parser CoreScDefn
pSc = pThen4 mkSc pVar (pZeroOrMore pVar) (pLit "=") pExpr
```

練習問題 1.20

`mkSc` を定義せよ

練習問題 1.21

関数適用と中置演算子適用の部分以外に対応した構文解析器を完成せよ

以下のプログラムを構文解析器のテストに使え

```
f = 3 ;
g x y = let z = y in z ;
h x = case (let y = x in y) of
        <1> -> 2
        <2> -> 5
```

1.22

「ぶらさがり `else` 問題」

`<2>` の選択肢は外側の `case` のものか、内側の `case` のものか

```
f x y = case x of
          <1> -> case y of
                   <1> -> 1;
          <2> -> 2
```

1.6.5 左再帰

適用式の生成規則は以下のようになる

$$expr \quad \rightarrow \quad expr\ aexpr$$

これをそのまま対応する構文解析器は

```
pExpr = pThen EAp pExpr pAexpr
```

となるが、これは停止しない。

文法の生成規則が左再帰を含まないように変更する

$$expr \quad \rightarrow \quad aexpr_1 \ \ldots \ aexpr_n \quad (n \geq 1)$$

対応する構文解析器は

```
pOneOrMore pAexpr `pApply` mkApChain
```

練習問題 1.23

```
mkApChain :: [CoreExpr] -> CoreExpr
```

を定義せよ。これを用いて、構文解析器が適用式を扱えるようにし、テストせよ

1.6.6 中置演算子の対応

$$
\begin{array}{lcll}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts & \\
 & | & \backslash\ var_1\ \ldots\ var_n\ \texttt{.}\ expr & (n \geq 1) \\
 & | & aexpr_1\ \ldots\ aexpr & (n \geq 1) \\
 & & & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 & \\
 & | & expr2 & \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 & \\
 & | & expr3 & \\
expr3 & \rightarrow & expr4\ relop\ expr4 & \\
 & | & expr4 & \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 & \\
 & | & expr5\ \texttt{-}\ expr5 & \\
 & | & expr5 & \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 & \\
 & | & expr6\ \texttt{/}\ expr6 & \\
 & | & expr6 & \\
expr6 & \rightarrow & aexpr_1\ \ldots\ aexpr_n & (n \geq 1)
\end{array}
$$

そのまま実装するとおそろしく効率が悪いので工夫が必要

$$
\begin{aligned}
expr1 &\rightarrow expr2\ expr1c \\
expr1c &\rightarrow \mid expr1 \\
&\mid \epsilon
\end{aligned}
$$

では、$expr1c$ に対応する構文解析器の型は？

```
data PartialExpr = NoOp ¦ FoundOp Name CoreExpr

pExpr1c :: Parser PartialExpr
pExpr1c = pThen FoundOp (pLit "¦") pExpr1 `pAlt` pEmpty NoOp

pExpr1 :: Parser CoreExpr
pExpr1 = pThen assembleOp pExpr2 pExpr1c

assembleOp :: CoreExpr -> PartialExpr -> CoreExpr
assembleOp e1 NoOp = e1
assembleOp e1 (BoundOp op e2) = EAp (EAp (EVar op) e1) e2
```

練習問題 1.24

- 文法を変形して、構文解析器を完成させよ
- 構文解析器をテストせよ