```
  root1 <- safeDiv ((-b) + discriminant) (2*a) -- division by zero?
  root2 <- safeDiv ((-b) - discriminant) (2*a)
  -- The monad ensures that we return a number only if no error has been raised
  return (root1,root2)
```

For a more detailed treatment of monads, and their relation to categorical monads, see the chapter on Category Theory and the chapter on Type Theory, where we will program with monads in Agda.

5.2 DE BRUIJN INDEXES

Nicolaas Govert **De Bruijn** proposed in [dB72] a way of defining λ-terms modulo α-conversion based on indices. The main goal of De Bruijn indices is to remove all variables from binders and replace every variable on the body of an expression with a number, called *index*, representing the number of λ-abstractions in scope between the ocurrence and its binder.

Consider the following example: the λ-term

$$\lambda x.(\lambda y.\ y(\lambda z.\ yz))(\lambda t.\lambda z.\ tx)$$

can be written with de Bruijn indices as

$$\lambda\ (\lambda(1\lambda(21))\ \lambda\lambda(23)\ ).$$

De Bruijn also proposed a notation for the λ-calculus changing the order of binders and λ-applications. A review on the syntax of this notation, its advantages and De Bruijn indexes, can be found in [Kam01]. In this section, we are going to describe De Bruijn indexes while preserving the usual notation of λ-terms; that is, the *De Bruijn indexes* and the *De Bruijn notation* are different concepts and we are going to use only the former.

Definition 28 (De Bruijn indexed terms)**.** We define recursively the set of λ-terms using de Bruijn notation following this BNF

$$\begin{aligned} \text{Exp} &::= \underbrace{\mathbb{N}}_{\textit{variable}} \mid \underbrace{(\lambda\ \text{Exp})}_{\textit{abstraction}} \mid \underbrace{(\text{Exp}\ \text{Exp})}_{\textit{application}} \\ \mathbb{N} &::= 0 \mid 1 \mid 2 \mid \ldots \end{aligned}$$

Our internal definition closely matches the formal one. The names of the constructors here are `Var`, `Lambda` and `App`:

```
-- | A lambda expression using DeBruijn indexes.
data Exp = Var Integer -- ^ integer indexing the variable.
         | Lambda Exp  -- ^ lambda abstraction
         | App Exp Exp -- ^ function application
         deriving (Eq, Ord)
```

This notation avoids the need for the Barendregt's variable convention and the α-reductions. It will be useful to implement λ-calculus without having to worry about the specific names of variables.

5.3 SUBSTITUTION

We define the substitution operation needed for the β-reduction on de Bruijn indices. In order to define the substitution of the n-th variable by a λ-term P on a given term, we must

- find all the ocurrences of the variable. At each level of scope we are looking for the successor of the number we were looking for before;
- decrease the higher variables to reflect the disappearance of a lambda;
- replace the ocurrences of the variables by the new term, taking into account that free variables must be increased to avoid them getting captured by the outermost lambda terms.

In our code, we apply `subs` to any expression. When it is applied to a λ-abstraction, the index and the free variables of the replaced term are increased with `incrementFreeVars`; whenever it is applied to a variable, the previous cases are taken into consideration.

```
-- | Substitutes an index for a lambda expression
subs :: Integer -> Exp -> Exp -> Exp
subs n p (Lambda e) = Lambda (subs (n+1) (incrementFreeVars 0 p) e)
subs n p (App f g)  = App (subs n p f) (subs n p g)
subs n p (Var m)
  | n == m    = p          -- The lambda is replaced directly
  | n <  m    = Var (m-1) -- A more exterior lambda decreases a number
  | otherwise = Var m      -- An unrelated variable remains untouched
```

Then β-reduction can be defined using this `subs` function.

```
betared :: Exp -> Exp
betared (App (Lambda e) x) = substitute 1 x e
betared e = e
```

5.4 DE BRUIJN-TERMS AND λ-TERMS

The internal language of the interpreter uses de Bruijn expressions, while the user interacts with it using lambda expressions with alphanumeric variables. Our definition of a λ-expression with variables will be used in parsing and output formatting.

```haskell
data NamedLambda = LambdaVariable String
                 | LambdaAbstraction String NamedLambda
                 | LambdaApplication NamedLambda NamedLambda
```

The translation from a natural λ-expression to de Bruijn notation is done using a dictionary which keeps track of the bounded variables

```haskell
tobruijn :: Map.Map String Integer -- ^ names of the variables used
         -> Context                -- ^ names already binded on the scope
         -> NamedLambda            -- ^ initial expression
         -> Exp
-- Every lambda abstraction is inserted in the variable dictionary,
-- and every number in the dictionary increases to reflect we are entering
-- a deeper context.
tobruijn d context (LambdaAbstraction c e) =
     Lambda $ tobruijn newdict context e
        where newdict = Map.insert c 1 (Map.map succ d)

-- Translation distributes over applications.
tobruijn d context (LambdaApplication f g) =
     App (tobruijn d context f) (tobruijn d context g)

-- We look for every variable on the local dictionary and the current scope.
tobruijn d context (LambdaVariable c) =
  case Map.lookup c d of
    Just n  -> Var n
    Nothing -> fromMaybe (Var 0) (MultiBimap.lookupR c context)
```

while the translation from a de Bruijn expression to a natural one is done considering an infinite list of possible variable names and keeping a list of currently-on-scope variables to name the indices.

```haskell
-- | An infinite list of all possible variable names
-- in lexicographical order.
variableNames :: [String]
variableNames = concatMap (MatchLowercasereplicateM ['a'..'z']) [1..]

-- | A function translating a deBruijn expression into a
-- natural lambda expression.
nameIndexes :: [String] -> [String] -> Exp -> NamedLambda
nameIndexes _    _    (Var 0) = LambdaVariable "undefined"
nameIndexes used _    (Var n) =
  LambdaVariable (used !! pred (fromInteger n))
nameIndexes used new (Lambda e) =
  LambdaAbstraction (head new) (nameIndexes (head new:used) (tail new) e)
```

```haskell
nameIndexes used new (App f g) =
  LambdaApplication (nameIndexes used new f) (nameIndexes used new g)
```

5.5 EVALUATION

As we proved on Corollary 1, the leftmost reduction strategy will find the normal form of any given term provided that it exists. Consequently, we will implement reduction in our interpreter using a function that simply applies the leftmost possible reductions at each step. As a side benefit, this will allow us to show how the interpreter performs step-by-step evaluations to the final user, as discussed in the verbose mode section.

```haskell
-- | Simplifies the expression recursively.
-- Applies only one parallel beta reduction at each step.
simplify :: Exp -> Exp
simplify (Lambda e)           = Lambda (simplify e)
simplify (App (Lambda f) x)   = betared (App (Lambda f) x)
simplify (App (Var e) x)      = App (Var e) (simplify x)
simplify (App a b)            = App (simplify a) (simplify b)
simplify (Var e)              = Var e

-- | Applies repeated simplification to the expression until it stabilizes and
-- returns all the intermediate results.
simplifySteps :: Exp -> [Exp]
simplifySteps e
  | e == s    = [e]
  | otherwise = e : simplifySteps s
  where s = simplify e
```

From the code we can see that the evaluation finishes whenever the expression stabilizes. This can happen in two different cases

- there are no more possible β-reductions, and the algorithm stops; or
- β-reductions do not change the expression. The computation would lead to an infinite loop, so it is immediately stopped. An common example of this is the λ-term $(\lambda x.xx)(\lambda x.xx)$.

5.6 PRINCIPAL TYPE INFERENCE

The interpreter implements the unification and type inference algorithms described in Lemma 4 and Theorem 5. Their recursive nature makes them very easy to implement directly on Haskell. We implement a simply-typed lambda calculus with Curry-style typing and type templates. Our type system has

- an unit type;
- a void type;
- product types;
- union types;
- and function types.

```haskell
-- | A type template is a free type variable or an arrow between two
-- types; that is, the function type.
data Type = Tvar Variable
          | Arrow Type Type
          | Times Type Type
          | Union Type Type
          | Unitty
          | Bottom
          deriving (Eq)
```

We will work with substitutions on type templates. They can be directly defined as functions from types to types. A basic substitution that inserts a given type on the place of a variable will be our building block for more complex ones.

```haskell
type Substitution = Type -> Type

-- | A basic substution. It changes a variable for a type
subs :: Variable -> Type -> Substitution
subs x typ (Tvar y)
  | x == y    = typ
  | otherwise = Tvar y
subs x typ (Arrow a b) = Arrow (subs x typ a) (subs x typ b)
subs x typ (Times a b) = Times (subs x typ a) (subs x typ b)
subs x typ (Union a b) = Union (subs x typ a) (subs x typ b)
subs _ _ Unitty = Unitty
subs _ _ Bottom = Bottom
```

Unification will be implemented making extensive use of the `Maybe` monad. If the unification fails, it will return an error value, and the error will be propagated to the whole computation. The algorithm is exactly the same that was defined in Lemma 4.

```haskell
-- | Unifies two types with their most general unifier. Returns the substitution
-- that transforms any of the types into the unifier.
unify :: Type -> Type -> Maybe Substitution
unify (Tvar x) (Tvar y)
  | x == y    = Just id
  | otherwise = Just (subs x (Tvar y))
unify (Tvar x) b
  | occurs x b = Nothing
```

```
  | otherwise  = Just (subs x b)
unify a (Tvar y)
  | occurs y a = Nothing
  | otherwise  = Just (subs y a)
unify (Arrow a b) (Arrow c d) = unifypair (a,b) (c,d)
unify (Times a b) (Times c d) = unifypair (a,b) (c,d)
unify (Union a b) (Union c d) = unifypair (a,b) (c,d)
unify Unitty Unitty = Just id
unify Bottom Bottom = Just id
unify _ _ = Nothing

-- | Unifies a pair of types
unifypair :: (Type,Type) -> (Type,Type) -> Maybe Substitution
unifypair (a,b) (c,d) = do
  p <- unify b d
  q <- unify (p a) (p c)
  return (q . p)
```

The type inference algorithm is more involved. It takes a list of fresh variables, a type context, a lambda expression and a constraint on the type, expressed as a type template. It outputs a substitution. As an example, the following code shows the type inference algorithm for function types.

```
-- | Type inference algorithm. Infers a type from a given context and expression
-- with a set of constraints represented by a unifier type. The result type must
-- be unifiable with this given type.
typeinfer :: [Variable] -- ^ List of fresh variables
          -> Context    -- ^ Type context
          -> Exp        -- ^ Lambda expression whose type has to be inferred
          -> Type       -- ^ Constraint
          -> Maybe Substitution

typeinfer (x:vars) ctx (App p q) b = do -- Writing inside the Maybe monad.
  sigma <- typeinfer (evens vars) ctx                  p (Arrow (Tvar x) b)
  tau   <- typeinfer (odds  vars) (applyctx sigma ctx) q (sigma (Tvar x))
  return (tau . sigma)
  where
    -- The list of fresh variables has to be split into two
    odds [] = []
    odds [_] = []
    odds (_:e:xs) = e : odds xs
    evens [] = []
    evens [e] = [e]
    evens (e:_:xs) = e : evens xs
```

The final form of the type inference algorithm will use a normalization algorithm shortening the type names and will apply the type inference to the empty type context. The complete code can be found on the . A generalized version of the type inference algorithm is used to generate derivation trees from terms, as it was described in Propositions as types.

In order to draw these diagrams in Unicode characters, a data type for character blocks has been defined. A monoidal structure is defined over them; blocks can be joined vertically and horizontally; and every deduction step can be drawn independently.

```
newtype Block = Block { getBlock :: [String] }
  deriving (Eq, Ord)

instance Monoid Block where
  mappend = joinBlocks -- monoid operation, joins blocks vertically
  mempty  = Block [[]] -- neutral element

-- Type signatures
joinBlocks :: Block -> Block -> Block
stackBlocks :: String -> Block -> Block -> Block
textBlock :: String -> Block
deductionBlock :: Block -> String -> [Block] -> Block
box :: Block -> Block
```

6

USER INTERACTION

6.1 MONADIC PARSER COMBINATORS

A common approach to building parsers in functional programming is to model parsers as functions. Higher-order functions on parsers act as *combinators*, which are used to implement complex parsers in a modular way from a set of primitive ones. In this setting, parsers exhibit a monad algebraic structure, which can be used to simplify the combination of parsers. A technical report on **monadic parser combinators** can be found on [HM96].

The use of monads for parsing was discussed for the first time in [Wad85], and later in [Wad90] and [HM98]. The parser type is defined as a function taking an input `String` and returning a list of pairs, representing a successful parse each. The first component of the pair is the parsed value and the second component is the remaining input. The Haskell code for this definition is

```haskell
-- A parser takes a string an returns a list of possible parsings with
-- their remaining string.
newtype Parser a = Parser (String -> [(a,String)])
parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p
-- A parser can be composed monadically, the composed parser (p >>= q)
-- applies q to every possible parsing of p. A trivial one is defined.
instance Monad Parser where
  return x = Parser (\s -> [(x,s)]) -- Trivial parser, directly returns x.
  p >>= q  = Parser (\s -> concat [parse (q x) s' | (x,s') <- parse p s ])
```

where the monadic structure is defined by `bind` and `return`. Given a value, the `return` function creates a parser that consumes no input and simply returns the given value. The »= function acts as a sequencing operator for parsers. It takes two parsers and applies the second one over the remaining inputs of the first one, using the parsed values on the first parsing as arguments.

An example of primitive **parser** is the `item` parser, which consumes a character from a non-empty string. It is written in Haskell code using pattern matching on the string as

```haskell
item :: Parser Char
item = Parser (\s -> case s of "" -> []; (c:s') -> [(c,s')])
```

and an example of **parser combinator** is the `many` function, which creates a parser that allows one or more applications of the given parser

```haskell
many :: Paser a -> Parser [a]
many p = do
  a  <- p
  as <- many p
  return (a:as)
```

in this example `many item` would be a parser consuming all characters from the input string.

6.2 PARSEC

Parsec is a monadic parser combinator Haskell library described in [Lei01]. We have chosen to use it due to its simplicity and extensive documentation. As we expect to use it to parse user live input, which will tend to be short, performance is not a critical concern. A high-performace library supporting incremental parsing, such as **Attoparsec** [O'S16], would be suitable otherwise.

6.3 VERBOSE MODE

As we explained previously on the evaluation section, the simplification can be analyzed step-by-step. The interpreter allows us to see the complete evaluation when the `verbose` mode is activated. To activate it, we can execute `:verbose on` in the interpreter.

The difference can be seen on the following example, which shows the execution of 1 + 2, first without intermediate results, and later, showing every intermediate step.

```
mikro> plus 1 2
λa.λb.(a (a (a b))) ⇒ 3

mikro> :verbose on
verbose: on
mikro> plus 1 2
((plus 1) 2)
((λλλ((4 2) ((3 2) 1)) λλ(2 1)) λλ(2 (2 1)))
(λλ((λλ(2 1) 2) ((3 2) 1)) λλ(2 (2 1)))
λλ((λλ(2 1) 2) ((λλ(2 (2 1)) 2) 1))
λλ(λ(3 1) (λ(3 (3 1)) 1))
```

```
λλ(2 (λ(3 (3 1)) 1))
λλ(2 (2 (2 1)))

λa.λb.(a (a (a b))) ⇒ 3
```

The interpreter output can be colored to show specifically where it is performing reductions. It is activated by default, but can be deactivated by executing `:color off`. The following code implements *verbose mode* in both cases.

```
-- | Shows an expression, coloring the next reduction if necessary
showReduction :: Exp -> String
showReduction (Lambda e)         = "λ" ++ showReduction e
showReduction (App (Lambda f) x) = betaColor (App (Lambda f) x)
showReduction (Var e)            = show e
showReduction (App rs x)         =
  "(" ++ showReduction rs ++ " " ++ showReduction x ++ ")"
showReduction e                  = show e
```

6.4 SKI MODE

Every λ-term can be written in terms of SKI combinators. SKI combinator expressions can be defined as a binary tree having S, K, and I as possible leafs.

```
data Ski = S | K | I | Comb Ski Ski | Cte String
```

The SKI-abstraction and bracket abstraction algorithms are implemented on Mikrokosmos, and they can be used by activating the *ski mode* with `:ski on`. When this mode is activated, every result is written in terms of SKI combinators.

```
mikro> 2
λa.λb.(a (a b)) ⇒ S(S(KS)K)I ⇒ 2
mikro> and
λa.λb.((a b) a) ⇒ SSK ⇒ and
```

The code implementing these algorithms follows directly from the theoretical version in [HS08].

```
-- | Bracket abstraction of a SKI term, as defined in Hindley-Seldin
-- (2.18).
bracketabs :: String -> Ski -> Ski
bracketabs x (Cte y) = if x == y then I else Comb K (Cte y)
bracketabs x (Comb u (Cte y))
  | freein x u && x == y = u
  | freein x u           = Comb K (Comb u (Cte y))
```