# Exercise 1 — assigned Thursday 1 February — due Tuesday 13 February

The project will be carried out in teams. The first task is to form and announce teams.

The implementation language is Haskell. More specifically, the project can be implemented in the Haskell 2010 standard language. If your team wishes to use GHC language extensions, you are free to do so; when you present your project, you will teach the class about any language extensions you have used.

The project will be carried out in several parts, called exercises.

For most parts of the project, you will be provided with some skeleton code in the form of type declarations, function signatures, or working components.

## 1.1 Core lambda language

We start the project with a small core lambda language, consisting of the lambda calculus with booleans and integers. Later we will add further syntactic forms.

The reference for this exercise is Pierce, *Types and Programming Languages*, Chapters 5 and 9.

Here is the concrete syntax in BNF:

```
Type -->
    arrow lpar Type comma Type rpar
  | Bool_keyword
  | Int_keyword

Term -->
    identifier
  | abs_keyword  lpar identifier colon Type fullstop Term rpar
  | app_keyword  lpar Term comma Term rpar
  | true_keyword
  | false_keyword
  | if_keyword  Term then_keyword  Term else_keyword  Term fi_keyword
  | intliteral
  | plus lpar Term comma Term rpar
  | minus lpar Term comma Term rpar
  | mul lpar Term comma Term rpar
  | div lpar Term comma Term rpar
  | nand lpar Term comma Term rpar
  | equal lpar Term comma Term rpar
```

```
   | lt lpar Term comma Term rpar
   | lpar Term rpar
```

Here is an informal description of the terminal symbols (tokens) used in the grammar above:

| | |
|---|---|
| `arrow` | `->` |
| `lpar` | `(` |
| `comma` | `,` |
| `rpar` | `)` |
| `Bool_keyword` | `Bool` |
| `Int_keyword` | `Int` |
| `identifier` | an identifier, e.g., as in Haskell |
| `abs_keyword` | `abs` |
| `colon` | `:` |
| `fullstop` | `.` |
| `app_keyword` | `app` |
| `true_keyword` | `true` |
| `false_keyword` | `false` |
| `if_keyword` | `if` |
| `then_keyword` | `then` |
| `else_keyword` | `else` |
| `fi_keyword` | `fi` |
| `intliteral` | a non-negative decimal numeral |
| `plus` | `+` |
| `minus` | `-` |
| `mul` | `*` |
| `div` | `/` |
| `nand` | `^` |
| `equal` | `=` |
| `lt` | `<` |

White space, such as space, tab, and newline characters, is permitted between tokens. White space is required between adjacent keyword tokens.

Comment syntax is as in Haskell.

Here are some example programs:

```
app (abs (x: Int        . 1234), 10)  -- would look better without the space
```

---

```
if true then true else false fi
   -- evaluates to true
```

---

```
if =(0,0) then 8 else 9 fi    -- evaluates to 8
```

---

```
{- a division example -}
/(4294967295,76)
```

## 1.2 Parsing

Write a parser for this language. (You may split this task into a scanner and a parser proper.)

## 1.3 Binding and free variables

Here you will define a handful of functions to manipulate the abstract syntax. Place them in the module *AbstractSyntax*.

Implement a function to enumerate the free variables of a term:

$$fv :: Term \to [Var]$$

Substitution: *subst x s t*, or in writing $[x \mapsto s]t$, is the result of substituting $s$ for the free occurrences of $x$ in $t$. Implement it.

$$subst :: Var \to Term \to Term \to Term$$

## 1.4 Arithmetic

The provided module *IntegerArithmetic* formalizes the primitive operators for integer arithmetic, which is 64-bit 2's complement, with silent overflow.

## 1.5 Type checker

It is always good to be sure a program is well-typed before we try to evaluate it. You can use the following type checker or write your own.

**module** *Typing* **where**
**import** *Data.Maybe*
**import** *Data.List*
**import** *qualified AbstractSyntax as S*
**data** *Context* = *Empty*

```
                | Bind Context S.Var S.Type
              deriving Eq
instance Show Context where
  show Empty        = "<>"
  show (Bind Γ x τ) = show Γ ++ "," ++ x ++ ":" ++ show τ
contextLookup :: S.Var → Context → Maybe S.Type
contextLookup x Empty = Nothing
contextLookup x (Bind Γ y τ)
  | x ≡ y      = Just τ
  | otherwise = contextLookup x Γ
typing :: Context → S.Term → Maybe S.Type
typing Γ t = case t of
  S.Var x → contextLookup x Γ
  S.Abs x τ1 t2 → do τ2 ← typing (Bind Γ x τ1) t2; Just (S.TypeArrow τ1 τ2)
  S.App t1 t2 → do S.TypeArrow τ11 τ12 ← typing Γ t1
    τ ← typing Γ t2
    if τ ≡ τ11 then Just τ12 else Nothing
  S.Tru → Just S.TypeBool
  S.Fls → Just S.TypeBool
  S.If t1 t2 t3 → do S.TypeBool ← typing Γ t1
    τ ← typing Γ t2
    tau' ← typing Γ t3
    if tau' ≡ τ then Just τ else Nothing
  S.IntConst _ → Just S.TypeInt
  S.IntAdd t1 t2 → arith t1 t2
  S.IntSub t1 t2 → arith t1 t2
  S.IntMul t1 t2 → arith t1 t2
  S.IntDiv t1 t2 → arith t1 t2
  S.IntNand t1 t2 → arith t1 t2
  S.IntEq t1 t2 → rel t1 t2
  S.IntLt t1 t2 → rel t1 t2
  where
    arith t1 t2 = do S.TypeInt ← typing Γ t1; S.TypeInt ← typing Γ t2; Just S.TypeInt
    rel t1 t2 = do S.TypeInt ← typing Γ t1; S.TypeInt ← typing Γ t2; Just S.TypeBool
typeCheck :: S.Term → S.Type
typeCheck t =
  case typing Empty t of
    Just τ → τ
    _ → error "type error"
```

## 1.6 Main program

Write a main program which will (1) read the program text from a file into a string, (2) invoke the parser to produce an abstract syntax tree for the program, (3) print the free variables of the program, (4) in case the program is closed (has no free variables), type-check the program, and (5) in case the program has a type, evaluate it with respect to its call-by-value structural operational semantics, as provided.

Can you produce informative error messages from the parser and the type checker? Can you display the proof of typing? Can you display the individual evaluation steps and their proofs?

## 1.7 Provided code

```
exercise1.lhs
AbstractSyntax.lhs
IntegerArithmetic.lhs
Typing.lhs
StructuralOperationalSemantics_CBV.lhs
Latex.lhs
```

## 1.8 What to submit

Prepare a project report (PDF). If you want, you can prepare the code and the report together in literate Haskell (like this assignment). Use Canvas to submit a tar file with all the code and the report.