Resit Advanced Programming (I00032)

July 8 2024

- This is a digital exam: **provide your answers in the file `ap24.txt`**. This file is in the folder (starting at "ThisPC") `Documents\Exam`. Do not move this file!
- In `Documents\Exam` you also find SaC and Clean modules to get you started with the exam questions. See instructions how to use them with the SaC (step 5) and Clean compiler (step 6).
- This exam is *closed book* (*gesloten boek*). Only this exam and the information provided on the machine can be used. This includes the SaC compiler and its documentation, as well as the Clean system.
- This exam consists of 5 assignments. The weight of the parts is indicated in the margin. You can obtain a maximum of 100 points. There is an appendix with the main types and combinators of iTask.
- Read the exam carefully. Do not hesitate to ask clarification (via the proctor) if an assignment is unclear.
- All functions and data-structures must be defined in correct SaC or Clean syntax.
- It is not required to hand in compiling and tested code. Show that you understand the concepts. Although you have compilers available, use this tooling wisely. Testing and debugging all code will probably cost you too much time.
- The workflow to use SaC and Clean looks like this:
  1. Start VSCode (e.g. via Windows Search: 

).
  2. In the "Terminal" menu, select "New Terminal".
  3. In the terminal, next to "powershell", click the pop-up icon next to "+" () and select "UbuntuSaC (WSL)".
  4. In the terminal, enter the command:
     `su - ap`
     A commandline prompt appears that starts with `ap@`.
  5. In this directory you can create a SaC program, say `main.sac`, compile it with `sac2c main.sac`, and execute it via `./a.out`. Warning: do not use the `sac2c` compiler flag `-check c`. You can use VSCode to open ("File" menu, command "Open File...") the SaC program by navigating to (starting at "ThisPC"):
     `Documents\Ubuntu\rootfs\home\ap\`.
  6. For Clean and iTask you work in the folder `itasks-template`, so, in the same terminal enter:
     `cd itasks-template`
     (a) In this directory you find the `nitrile.yml` file in which you set which Clean module to compile and where the executable is generated. It is currently set to the example iTask module `HelloWorld.icl` that you can find in the directory `scr`, and the executable is generated as `bin/HelloWorld`.
     (b) You can use VSCode to open ("File" menu, command "Open File...") a Clean program by navigating to (starting at "ThisPC"):
     `Documents\Ubuntu\rootfs\home\ap\itasks-template\src\`.
     (c) In the `itasks-template` directory, compile your program with `nitrile build` and execute it by `./bin/HelloWorld`.
     (d) To test your running iTask application, open a browser and navigate to `localhost:8080`. Do not forget to terminate the running program before recompiling an edited version.
     (e) Remember that you are not connected to the internet. Hence, things like `nitrile fetch` will not work and can harm your project.
  7. If you use VSCode to create a new file, then the file permissions need to be set properly to compile it. Save and close the file in VSCode. In the terminal enter:
     `chmod 644 filename`
     You can now open and edit the file in VSCode and compile it.

1 Array Programming

1.a) SaC supports a built-in operation `_take_SxV_` that takes two arguments, a scalar integer and a vector of arbitrary element type. It returns a potentially smaller vector containing the first few elements of the given vector. If the scalar value is a non-negative value that does not exceed the length of the vector, the length of the result is determined by the scalar. Otherwise, its behaviour is undefined. 5 pt.

Modify the following function definition using *type pattern* to express the domain constraints between arguments and return values. You may use existing functionality from the standard library to express any required constraints on argument values.

```
float[.] mytake (int s, int[.] a)
{
  return _take_SxV_ (s, a);
}
```

Solution:

```
float[s] mytake (int s, int[n] a) | (s >= 0) && (s <= n)
{
  return _take_SxV_ (s, a);
}
```

1.b) Write a different version of `mytake` that behaves in the same way as the previous one when being applied to an integer scalar and an integer vector, but which also accepts higher-dimensional arrays of integers as second argument. In those cases, it should return scalar-many hyper-planes of the second argument. You may **not(!)** use `take` or `drop` from the standard library! 5 pt.

For example, `mytake (2, [[0,1], [2,3], [4,5]])` should result in `[[0,1],[2,3]]`. Again, make sure you express all argument constraints as well as return shape relations precisely.

Solution:

```
float[s,m:shp] mytake (int s, float[n,m:shp] a) | (s >= 0) && (s <= n)
{
  return {[i] -> a[i] | [i] < [s]};
}
```

1.c) Define a function `concat` that takes two integer vectors and appends them. You are **not(!)** allowed to use ++ from the standard library! Make sure you express all constraints on the signature precisely. 5 pt.

Solution:

```
int[o] concat (int[m] a, int[n] b) | o == m+n
{
   return {[i] -> a[i]   | [i] < [m];
           [i] -> b[i-m] | [m] <= [i] < [m+n]};
}
```

1.d) Overload your definition of `concat` so that you allow for higher-dimensional arrays as arguments, provided that all shape components of the two argument arrays that precede the last dimension are identical. 5 pt.

A few examples:

```
concat ([1,2],           [3,4,5])            ==  [1,2,3,4,5]
concat ([[[1,2,3]]],     [[[4,5]]])          ==  [[[1,2,3,4,5]]]
concat ([[1,2],[3,4]],   [[5,6,7],[8,9,10]]) ==  [[1,2,5,6,7],[3,4,8,9,10]]
```

Solution:

```
int[s:shp,o] concat (int[s:shp,m] a, int[s:shp,n] b) | o == m+n
{
    return {iv -> concat (a[iv], b[iv]) | iv < shp};
}
```

1.e) Define a different version `t_concat`, which concatenates the first axis rather than the last axis of two n-dimensional arrays. 5 pt.

We should have:

```
t_concat ([1,2],               [3,4,5])          ==  [1,2,3,4,5]
t_concat ([[1,2],[3,4]],       [[5,6],[7,8]])    ==  [[1,2],[3,4],[5,6],[7,8]]
t_concat ([[1,2],[3,4],[5,6]], [[7,8],[9,10]])   ==  [[1,2],[3,4],[5,6],[7,8],[9,10]]
```

Again, make sure you capture all argument and result constraints in the signature of your function definition.

Solution:

```
int[o,s:shp] t_concat (int[m,s:shp] a, int[n,s:shp] b)
{
    return transpose (concat (transpose(a), transpose (b)));
}
```

Please check that you have copy-pasted your solutions in “ap24.txt”.

2 Concurrency Pattern in SaC

Consider the following SaC code:

```
int[n:shp] add (int[n:shp] a, int[n:shp] b)
{
    return {iv -> a[iv] + b[iv]};
}

int main()
{
    a = reshape ([2,4], iota( 8));
    print (add (a, a));
    return 0;
}
```

2.a) Which concurrency pattern below does the call to `add` exhibit? 5 pt.

```
       pattern I
           |
     ------------
     |          |
  ------     ------
  | | | |    | | | |
  ------     ------
     |          |
     ------------
           |
```

or

```
     pattern II
         |
  ---------------
  | | | | | | | |
  ---------------
         |
```

Solution:
The solution above implements pattern II.

2.b) Re-implement `add` so that it implements the other pattern! 10 pt.

Solution:

```
int[m,n:shp] add (int[m,n:shp] a, int[m,n:shp] b)
{
    return {[i] -> add (a[i], b[i]) | [i] < [m]};
}

int add (int a, int b)
{
    return a + b;
}
```

2.c) Which of the two patterns do you expect to be more efficient, assuming we would run it on a much larger array? Provide arguments based on the concurrency pattern. 10 pt.

Solution:
The second pattern provides wider concurrency and requires fewer synchronisation. Therefore, it should perform better.

Please check that you have copy-pasted your solutions in “ap24.txt”.

3 WHILE + I/O

In this and the next assignment we work with the WHILE programming language and extend it with I/O operations to read an integer value from the console and to print a string to console (these are statements) and build string values and distinguish between integer and string variables (these are expressions). Appendix A presents the extended language. An initial implementation module `WHILE.icl` is available to get you started. You still need to copy your answers to the answer file as instructed in the preamble of this exam text.

For the evaluation of WHILE+I/O style programs, we introduce the following data types:

```
:: *State   = { world :: *World
              , vars  :: 'Data.Map'.Map Var Dynamic
              }
:: Var     :== String
:: Result a = Result a | Error String
:: Eval   a =: E (State → *(Result a,State))
```

Evaluation of expressions and statements can fail, which is captured with the `Result` type: a successful evaluation with value `x` produces `(Result x)`, whereas a failing evaluation yields an error message `msg` as `(Error msg)`.

`State` uses the efficient key-value pair implementation `Data.Map` to associate the value, stored as a `Dynamic`, of a variable, represented with the type `Var`. The `*World` is required to access the console. The following operations are given:

```
write :: Var a  → Eval () | TC a   // write v x associates value x with variable v
read  :: Var    → Eval a  | TC a   // read v retrieves the currently associated value of variable v (this operation may fail)
cin   ::          Eval Int         // cin reads the next integer input from the console
cout  :: String → Eval ()          // cout s writes s to the console
```

3.a) Implement the instances for the monadic type constructor classes, such that they adhere to the usual laws (you do not have to prove this): 5 pt.

```
instance pure      Eval where ...
instance Monad     Eval where ...
instance Functor   Eval where ...
instance <*>       Eval where ...
instance MonadFail Eval where ...
```

Solution:

```
instance pure      Eval where pure a        = E (λs = (Result a, s))
instance Monad     Eval where bind (E f) g = E λs = case f s of
                                                 (Result x,  t) = let (E h) = g x in h t
                                                 (Error msg, t) = (Error msg, t)
instance Functor   Eval where fmap f (E g) = E λs = case g s of
                                                 (Result x, t) = (Result (f x),t)
                                                 (Error msg,t) = (Error msg, t)
instance <*>       Eval where (<*>) f x     = f >>=λg = x >>=pure o g
instance MonadFail Eval where fail msg      = E λs = (Error msg,s)
```

3.b) Implement a monadic evaluator for expressions that deals with errors. It must have the following signature: 5 pt.

```
evalE :: (Expr a) → Eval a | TC a
```

Solution:

```
evalE :: (Expr a) → Eval a | TC a
evalE e
  = case e of
        Lit     a   = pure a
```

```
Add    bm x y = evalE x >>= λvx = evalE y >>= λvy = pure (bm.ba (vx + vy))
Mul    bm x y = evalE x >>= λvx = evalE y >>= λvy = pure (bm.ba (vx * vy))
Div    bm x y = evalE x >>= λvx = evalE y >>= λvy = if (vy == 0)
                                                     (fail "division by zero")
                                                     (pure (bm.ba (vx / vy)))
Leq    bm x y = evalE x >>= λvx = evalE y >>= λvy = pure (bm.ba (vx ≤ vy))
Not    bm x   = evalE x >>= λvx = pure (bm.ba (not vx))
Concat bm x y = evalE x >>= λvx = evalE y >>= λvy = let str1 = toString vx
                                                        str2 = toString vy
                                                    in (pure (bm.ba (str1 ++ str2)))
VarI   bm v   = read v >>= λvv = pure (bm.ba vv)
VarS   bm v   = read v >>= λvv = pure (bm.ba vv)
```

3.c) Implement a monadic evaluator for statements `Stmt`. It must have the following signature: 5 pt.

```
evalS :: Stmt → Eval ()
```

Solution:

```
evalS :: Stmt → Eval ()
evalS stmt
  = case stmt of
      v =. e      = evalE e >>= λx = write v x
      s1 :. s2    = evalS s1 >>= λ_ = evalS s2
      If c t e    = evalE c >>= λb = if b (evalS t) (evalS e)
      While c b   = evalS (If c (b :. stmt) Skip)
      Skip        = pure ()
      Cin v       = cin >>= λx = write v x
      Cout e      = evalE e >>= λvs = cout vs
```

Please check that you have copy-pasted your solutions in “ap24.txt”.

4 Shallow and Tagless DSL

In this assignment we improve the WHILE DSL by enforcing type safe variables and offering multiple views. We introduce a new type `Var`` for variables and add an operation to the `Eval`uator to obtain a fresh variable name:

```
:: Var` a :== String

fresh :: Eval (Var` a)
fresh = E λs=:{vars} = (Result ("v" <+ 'Data.Map'.mapSize vars), s)
```

In Appendix B you find the type constructor class definitions of the improved WHILE language.

4.a) Define the instances of `Eval` for the involved type constructor classes: 8 pt.

```
instance expr Eval
   where ...
instance vars Eval a | TC a
   where ...
instance stmt Eval
   where ...
```

Solution:

```
instance expr Eval
   where lit x     = pure x
         add e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 + v2)
         mul e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 * v2)
         div e1 e2 = e1 >>= λv1 = e2 >>= λv2 = if (v2 == 0) (fail "division by zero") (pure (v1 / v2))
         leq e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 ≤ v2)
         neg e     = e  >>= λvb = pure (not vb)
         con e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure ("" <+ v1 <+ v2)
instance vars Eval a | TC a
   where var v = read v
         def f     = fresh >>= λv = let (a In e) = f v in a >>= λva = write v va >>| e
         (=.) v e  = e >>= λve = write v ve >>= λ_ = pure ve
instance stmt Eval
   where (.:.)  s t = s >>= λ_ = t
         if`  c t e = c >>= λb = if b t e
         while` c b = if` c (b .:. while` c b) skip`
         skip`      = pure ()
         cin`  v    = cin >>= λx = write v x
         cout` e    = e >>= λs = cout s
```

4.b) For printing we introduce the following data types and auxiliary functions:

```
:: Print a =: P (PS → *(a,PS))
:: *PS     = {i      :: Int         // generate fresh variable name
             ,indent :: Int         // current indentation level
             ,world  :: *World      // world for printing
             }

print    :: a → Print b | toString a   // print s writes s to the console
printNL  :: Print a                    // print a newline and the current indentation level to the console
inc      :: Print a                    // increment the indentation level
dec      :: Print a                    // decrement the indentation level
freshVar :: Print (Var` a)             // extract a fresh variable name
```

Implement the instances for the monadic type constructor classes, such that they adhere to the usual laws (you do not have to prove this): 4 pt.

```
instance pure    Print where ...
instance Monad   Print where ...
instance Functor Print where ...
instance <*>     Print where ...
```

> Solution:
>
> ```
> instance pure    Print where pure a      = P λs = (a, s)
> instance Monad   Print where bind (P f) g = P λs = let (x,t) = f s
>                                                       (P h) = g x
>                                                   in h t
> instance Functor Print where fmap f g    = g >>= pure ∘ f
> instance <*>     Print where (<*>) f x   = f >>= λg = x >>= pure ∘ g
> ```

4.c) Use these functions to give an implementation of `Print` to the type constructor classes, such that the program prints itself to the console. 8 pt.

```
instance expr Print
   where ...
instance vars Print a | TC a
   where ...
instance stmt Print
   where ...
```

> Solution:
>
> ```
> instance expr Print
>    where lit x     = print (show x)
>          add e1 e2 = print "(" >>| e1 >>| print " + " >>| e2 >>| print ")"
>          mul e1 e2 = print "(" >>| e1 >>| print " - " >>| e2 >>| print ")"
>          div e1 e2 = print "(" >>| e1 >>| print " / " >>| e2 >>| print ")"
>          leq e1 e2 = print "(" >>| e1 >>| print " ≤ " >>| e2 >>| print ")"
>          neg e     = print "(not " >>| e >>| print ")"
>          con e1 e2 = print "(" >>| e1 >>| print " ++ " >>| e2 >>| print ")"
> instance vars Print a | TC a
>    where var v     = print v
>          def f     = freshVar >>= λv =
>                        let (a In e) = f v
>                         in print "def " >>| print v >>| print " = " >>| a >>|
>                            print " In " >>| printNL >>| e
>          (=.) v e  = print v >>| print " =. " >>| e
> instance stmt Print
>    where (.:.)  s t = s >>| print ";" >>| printNL >>| t
>          if`  c t e = print "if " >>| c >>| print "{" >>| inc >>| printNL >>|
>                        t >>| dec >>| print "} else {" >>| inc >>|
>                        e >>| dec >>| print "}"
>          while` c b = print "while " >>| c >>| print "{" >>| inc >>| printNL >>|
>                        b >>| dec >>| printNL >>| print "}"
>          skip`      = print "skip"
>          cin`  v    = print "cin >> " >>| print v
>          cout` e    = print "cout << " >>| e >>| pure ()
> ```

Please check that you have copy-pasted your solutions in “ap24.txt”.

5 Task Oriented Programming: Nim

In this assignment we create a couple of tasks to allow two people to play the game of *Nim*. In a game of Nim, there is a number of (at least one) piles of objects (at least one in every pile). Pile numbering starts at one. The players A and B take turns, player A starts. At every move, a player selects a non-empty pile and removes at least one object and at most all objects from the selected pile. The game ends as soon as all piles are empty. The game is lost by the player who took away the last object(s). A game of Nim is modeled with the following data types, functions, and shared data source:

```
:: Nim    = {turn  :: Player    // the player who is allowed to play
            ,piles :: [Int]     // the current piles in the game
            }
:: Player = A | B               // player token
:: Move   = {pile   :: Int      // indicate a non-empty pile from Nim (count from 1)
            ,number :: Int      // indicate a positive number from selected pile, bounded by the number of its objects
            }
derive class iTask Nim, Player, Move
instance == Player where (==) p q = p === q

next :: Player -> Player        // the next player after a legal move
next A = B
next B = A

nimSDS :: SimpleSDSLens Nim     // the shared data source that holds the current Nim game state
nimSDS = sharedStore "nim" {turn = A, piles = [1..3]++[4,3..1]}

validMove :: Nim Move -> Bool   // a valid move identifies a valid pile and a valid number of objects
validMove nim=:{piles} move=:{pile,number}
  = 0 < pile && pile <= no_of_piles && no_at_pile > 0 && number <= no_at_pile && number > 0
where no_of_piles = length piles
      no_at_pile  = piles!!(pile-1)

gameOver :: Nim -> Bool         // the game is over when all piles are empty
gameOver nim = sum nim.Nim.piles == 0

selectPlayers :: Task (User,User)   // select two different users to play a game of Nim
selectPlayers = enterMultipleChoiceWithShared [] users >>*
                  [OnAction (Action "Ok") (ifValue (\choice = length choice == 2) (\[a,b:_] = return (a,b)))]
```

An initial implementation module `TOP.icl` is available to get you started. You still need to copy your answers to the answer file as instructed in the preamble of this exam text.

5.a) Implement the task function: 5 pt.

```
enterMove :: Nim -> Task Move
```

`(enterMove nim)` lets the player enter a `Move` value that is returned as a stable value only if it is valid (`validMove`).

Solution:

```
enterMove :: Nim -> Task Move
enterMove nim
  = enterInformation [] >>* [OnAction (Action "Ok") (ifValue (validMove nim) return)]
```

5.b) Implement the task function: 5 pt.

```
play :: Player -> Task Player
```

`(play p)` waits until `nimSDS` indicates that `p` must play. If the game is over, the task returns `p` as a stable value; otherwise, the player enters a move and updates `nimSDS` with that move, and continues recursively.

Solution:

```
play :: Player → Task Player
play me
  = wait (\nim = nim.Nim.turn == me) nimSDS >>- \nim =
    if (gameOver nim) (return me)
       (enterMove nim >>- λmove =
        set (applyMove move nim) nimSDS >-|
        play me
       )

applyMove :: Move Nim → Nim
applyMove move=:{pile,number} nim=:{turn,piles}
  = {nim & turn  = next turn
         , piles = [if (i == pile) (pile - number) pile \\ pile←piles & i←[1..]]
    }
```

5.c) Implement the task function: 5 pt.

```
game :: Task Player
```

game first selects two users to play a game of Nim. One of the users plays as player A, and the other as player B. Each player performs the play task and views nimSDS at the same time. As soon as one of the players returns a stable task value declaring to have won the game, this winner is shown.

Solution:

```
game :: Task Player
game = selectPlayers >>- λ(a,b) =
       ((a @: (play A -|| viewSharedInformation [] nimSDS))
         -||-
        (b @: (play B -|| viewSharedInformation [] nimSDS))
       ) >>- λwinner = viewInformation [] winner <<@ Hint "The winner is"
```

Please check that you have copy-pasted your solutions in “ap24.txt”.

A While+I/O deep GADT

In this appendix you find the deeply embedded representation of the WHILE language with the proposed extensions.

```
:: Expr a
    = Lit a                                              // Lit x: x is an int, bool, or string literal
    | Add  (BM a Int) (Expr Int) (Expr Int)              // Add bm a b: add value of a to value of b
    | Mul  (BM a Int) (Expr Int) (Expr Int)              // Mul bm a b: multiply value of a with value of b
    | Div  (BM a Int) (Expr Int) (Expr Int)              // Div bm a b: divide value of a by value of b (division by zero fails)
    | ∃b: Leq (BM a Bool) (Expr b) (Expr b)              // Leq bm a b: compare value of a with value of b with ≤
        & <, TC b
    | Not  (BM a Bool) (Expr Bool)                       // Not bm b: negate value of b
    | ∃b c: Concat (BM a String) (Expr b) (Expr c)       // Concat bm a b: concatenate string of a with string of b
        & TC, toString b & TC, toString c
    | VarI (BM a Int)    Var                             // VarI bm v: v represents an int variable
    | VarS (BM a String) Var                             // VarS bm v: v represents a string variable
:: Stmt
    = ∃a: (=.) infix 2 Var (Expr a) & TC a               // v =. e: assign value of e to v
    | (:.) infixr 1 Stmt Stmt                            // s :. t: execute s before t
    | If (Expr Bool) Stmt Stmt                           // If c t e: if c evaluates to True execute t, else e
    | While (Expr Bool) Stmt                             // While c s: while c evaluates to True execute s
    | Skip                                               // Skip: no operation
    | Cin  Var                                           // Cin v: get an Int from console and assign it to v
    | Cout (Expr String)                                 // Cout e: print string value of e to console

:: BM a b = {ab :: a → b, ba :: b → a}
bm = {ab = id, ba = id}
```

A program that asks for a positive number and computes and displays the factorial can look like this:

```
factorial :: Stmt
factorial
  = "a" =. Lit 0 :.                                          // int a = 0;
     While (Leq bm (VarI bm "a") (Lit 0)) (                  // while (a ≤ 0) {
        Cout (Lit "Please enter a positive number\n") :.     //     cout << “Please enter a positive number\n”;
        Cin "a"                                              //     cin >> a;
     ) :.                                                    // };
     "s" =. Concat bm (Lit "fac (")                          // s = concat (“fac (”,concat (a, “) = ”));
               (Concat bm (VarI bm "a") (Lit ") = ")) :.
     "x" =. Lit 1 :.                                         // int x = 1;
     While (Leq bm (Lit 1) (VarI bm "a")) (                  // while (1 ≤ a) {
        "x" =. Mul bm (VarI bm "x") (VarI bm "a") :.         //     x = x * a;
        "a" =. Add bm (VarI bm "a") (Lit -1)                 //     a = a + -1;
     ) :.                                                    // };
     "s" =. Concat bm (VarS bm "s") (VarI bm "x") :.         // s = concat (s, x);
     Cout (VarS bm "s") :.                                   // cout << s;
     Cout (Lit "\n")                                         // cout << “\n”;
```

B While + I/O shallow

In this appendix you find the shallow, overloaded, embedded representation of the WHILE language with the proposed extensions.

```
class expr v
where lit          :: a → v a | show a
      add          :: (v Int) (v Int) → v Int
      mul          :: (v Int) (v Int) → v Int
      div          :: (v Int) (v Int) → v Int
      leq          :: (v a) (v a) → v Bool | < a
      neg          :: (v Bool) → v Bool
      con          :: (v a) (v b) → v String | toString a & toString b
class vars v a
where var          :: (Var` a) → v a
      def          :: ((Var` a) → In (v a) (v b)) → v b
      (.=.) infixr 2 :: (Var` a) (v a) → v a
class stmt v
where (.:.) infixr 1 :: (v a) (v b) → v b
      if`          :: (v Bool) (v a) (v a) → v a
      while`       :: (v Bool) (v a) → v ()
      skip`        :: v ()
      cin`         :: (Var` Int) → v ()
      cout`        :: (v String) → v ()

:: In a b = In infix 0 a b

class show a :: a → String
instance show Int    // show argument as a number
instance show Bool   // show argument as True or False
instance show Char   // show newline as "\n", tab as "\t", others as the char
instance show String // show escape characters and delimit string with " and "
```

The shallow, overloaded version of the program of Appendix A looks like this:

```
factorial`
  = def λa = lit 0 In
    def λs = lit "" In
    while` (leq (var a) (lit 0)) (
        cout` (lit "Please enter a positive number\n") .:.
        cin` a
    ) .:.
    s .=. con (var s) (con (lit "fac (") (con (var a) (lit ") = "))) .:.
    def λx = lit 1 In
    while` (leq (lit 1) (var a)) (
        x .=. mul (var x) (var a) .:.
        a .=. add (var a) (lit -1)
    ) .:.
    s .=. con (var s) (var x) .:.
    cout` (var s) .:.
    cout` (lit "\n")
```

C Clean

```
// Monad and friends. Import these definitions as
import Control.Monad, Control.Monad.Fail, Control.Applicative, Data.Functor
class pure :: a → f a
class Functor f
where fmap :: (a → b) (f a) → f b
      (<$>) infixl 4 :: (a → b) (f a) → f b | Functor f
      (<$>) f fa :== fmap f fa
class (<*>) infixl 4 :: (f (a → b)) (f a) → f b
class Monad m | Applicative m
where bind :: !(m a) (a → m b) → m b
      (>>=) infixl 1 :: (m a) (a → m b) → m b | Monad m
      (>>=) ma a2mb :== bind ma a2mb
      (>>|) infixl 1 :: (m a) (m b) → m b | Monad m
      (>>|) ma mb :== ma >>= (λ_ → mb)
class MonadFail m | Monad m
where fail :: String → m a
// Map. Import this qualified to avoid name clashes.
import qualified Data.Map
newMap  :: Map k v
toList  :: (Map k v) → [(k,v)]
mapSize :: (Map k v) → Int
put     :: k v (Map k v) → Map k v | < k
get     :: k (Map k v) → ?v | < k
del     :: k (Map k a) → Map k a | < k
:: ? a = ?None | ?Just a // maybe
```

D iTask

```
:: Task a
return                   :: a → Task a                // task with stable value @1
(@) infixl 1             :: (Task a) (a → b) → Task b // map @2 to ((un)stable) task value of @1
// editor tasks (user creates / views / updates task value):
enterInformation         :: [EnterOption  m]   → Task m | iTask m
viewInformation          :: [ViewOption   m] m → Task m | iTask m
updateInformation        :: [UpdateOption m] m → Task m | iTask m
// editor tasks on SDS's (user views / updates SDS value):
viewSharedInformation   :: [ViewOption         r]   (sds () r w) → Task r | iTask r & T   C w & RWShared sds
updateSharedInformation :: [UpdateSharedOption r w] (sds () r w) → Task r | iTask r & iTask w & RWShared sds
// editor task customization (partial):
:: EnterOption        a   = ∃v: EnterAs        (v → a)              & iTask v
:: ViewOption         a   = ∃v: ViewAs         (a → v)              & iTask v
:: UpdateOption       a   = ∃v: UpdateAs       (a → v)  (a v → a)   & iTask v
:: UpdateSharedOption a b = ∃v: UpdateSharedAs (a → v)  (a v → b)   & iTask v
// parallel task combinators:
allTasks         :: [Task a]          → Task [a]    | iTask a            // collect all task values, is stable when all are stable
anyTask          :: [Task a]          → Task a      | iTask a            // collect first stable task value, unstable otherwise
(-||-) infixr 3  :: (Task a) (Task a) → Task a      | iTask a            // collect first stable task value, unstable otherwise
(-||)  infixl 3  :: (Task a) (Task b) → Task a      | iTask a & iTask b  // perform both, but collect @1
(||-)  infixr 3  :: (Task a) (Task b) → Task b      | iTask a & iTask b  // perform both, but collect @2
(-&&-) infixr 4  :: (Task a) (Task b) → Task (a, b) | iTask a & iTask b  // collect both task values, stable if both stable
// sequential task combinators:
(>-|)  infixl 1 :: (Task a)        (Task b) → Task b | iTask a & iTask b   // @2 after @1 has stable task value
(>>-)  infixl 1 :: (Task a) (a → Task b) → Task b | TC, JSONEncode{|*|} a // @2 after @1 has stable task value
(>>!)  infixl 1 :: (Task a) (a → Task b) → Task b | TC, JSONEncode{|*|} a // @2 after @1 has stable task value and user ok
(>>?)  infixl 1 :: (Task a) (a → Task b) → Task b | TC, JSONEncode{|*|} a // @2 after @1 has stable task value or user ok
(>?|)  infixl 1 :: (Task a)        (Task b) → Task b | TC, JSONEncode{|*|} a // @2 after @1 has stable task value or user ok
```

```
// general sequential task combinator:
(>>*)  infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | TC, JSONEncode{|*|} a
:: TaskCont a b = OnValue        ((TaskValue a) → ?b)  // continue on task value
                | OnAction Action ((TaskValue a) → ?b) // continue on user ok and task value
:: Action       = Action String
:: TaskValue a  = NoValue
                | Value a Stability
:: Stability :== Bool
// wrappers to create sequential task continuations (OnValue / OnAction):
always     :: b                          (TaskValue a) → ?b // all task values ok
never      :: b                          (TaskValue a) → ?b // no task value ok
hasValue   :: (a → b)                    (TaskValue a) → ?b // map @1 to (un)stable task value
ifStable   :: (a → b)                    (TaskValue a) → ?b // map @1 to stable task value only
ifUnstable :: (a → b)                    (TaskValue a) → ?b // map @1 to unstable task value only
ifValue    :: (a → Bool)  (a → b)        (TaskValue a) → ?b // map @2 to (un)stable task value only if @1 is true
ifCond     :: Bool b                     (TaskValue a) → ?b // @2 if @1
// shared data sources (global scope vs local scope):
:: SimpleSDSLens a :== SDSLens () a a
sharedStore :: String a → SimpleSDSLens a | JSONEncode{|*|}, JSONDecode{|*|}, TC a
withShared :: b ((SimpleSDSLens b) → Task a) → Task a | iTask a & iTask b
// atomic access to shared data source
get   ::              (sds () a w) → Task a | TC    a & TC w & Readable  sds
set   :: a            (sds () r a) → Task a | TC    a & TC r & Writeable sds
upd   :: (r → w)      (sds () r w) → Task w | TC    r & TC w & RWShared  sds
// transform shared data source into task (watch) or wait for first value with predicate (wait):
watch ::              (sds () r w) → Task r | TC    r & TC w & Readable, Registrable sds
wait  :: (r → Bool)   (sds () r w) → Task r | iTask r & TC w & RWShared  sds
// compose shared data sources (class constraints omitted to avoid clutter)
(>*<) infixl 6 :: (sds1 p rx wx) (sds2 p ry wy) → SDSParallel p (rx,ry) (wx,wy) | ...
(>*|) infixl 6 :: (sds1 p rx wx) (sds2 p ry wy) → SDSParallel p (rx,ry) wx      | ...
(|*<) infixl 6 :: (sds1 p rx wx) (sds2 p ry wy) → SDSParallel p (rx,ry) wy      | ...
(|*|) infixl 6 :: (sds1 p rx wx) (sds2 p ry wy) → SDSParallel p (rx,ry) ()      | ...
// globally accessible shared data sources (time, date, user, users)
currentTime :: SDSLens () Time ()
currentDate :: SDSLens () Date ()
currentUser :: SDSLens () User User
users       :: SDSLens () [User] ()
// task distribution (simplified):
(@:) infix 3 :: User (Task a) → Task a | iTask a // user @1 performs task @2
```