



# Exam Advanced Programming (I00032)

June 17 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 `src`, 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.

# 1 Array Programming

- 1.a) SaC supports a built-in selection operation `_sel_VxA_` which takes two arguments, an integer vector and an array of arbitrary element type. Provided the vector length matches the dimensionality of the array and the elements of the integer vector are legal indices into the array, it returns the corresponding scalar value. Modify the following function definition using *type pattern* to express the argument constraints precisely. You may use existing functionality from the standard library to express the required constraints. 5 pt.

```
float select (int[.] v, float[*] a)
{
    return _sel_VxA_ (v, a);
}
```

Solution:

```
float select (int[n] v, float[n:shp] a) | all ((0 <= v) && (v < shp))
{
    return _sel_VxA_ (v, a);
}
```

- 1.b) Write a different version of `select` that allows for shorter index vectors and returns the corresponding hyperplanes. For example, `select ([1], [[0,1], [2,3]])` should result in `[2,3]`. Again, make sure you express all argument constraints precisely. 5 pt.

Solution:

```
float[m:shp2] select (int[n] v, float[n:shp,m:shp2] a) | all ((0 <= v) && (v < shp))
{
    return {iv -> _sel_VxA_ (v ++ iv, a) | iv < shp2};
}
```

- 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) Extend your definition of `concat` so that you allow for higher-dimensional arrays as arguments, provided that all shape components of the two arrays that follow the first dimension are identical. For example, `concat ([[1,2,3]], [[4,5,6], [7,8,9]])` should yield `[[1,2,3], [4,5,6], [7,8,9]]` 5 pt.

Solution:

```
int[o,s:shp] concat (int[m,s:shp] a, int[n,s:shp] b) | o == m+n
{

```

```

    return {[i] -> a[i] | [i] < [m];
            [i] -> b[i-m] | [m] <= [i] < [m+n]};
}

```

- 1.e) Create a ranked version `r_concat`, which takes a scalar integer as additional first argument. This argument should indicate which dimension is to be concatenated. We should have: `r_concat (0,a,b) == concat (a,b)` for any set of legal arguments `a` and `b` of `concat`. 5 pt.

As another example, consider `r_concat (1, [[1,2],[3,4]], [[5,6,7],[8,9,10]])` which should result in `[[1,2,5,6,7],[3,4,8,9,10]]`.

Solution:

```

int[r:shp,o,s:shp2] r_concat (int r, int[r:shp,m,s:shp2] a, int[r:shp,n,s:shp2] b)
| o == m+n
{
    return {iv -> concat (a[iv] + b[iv]) | iv < shp};
}

```

---

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

---

## 2 Concurrency Pattern in SaC

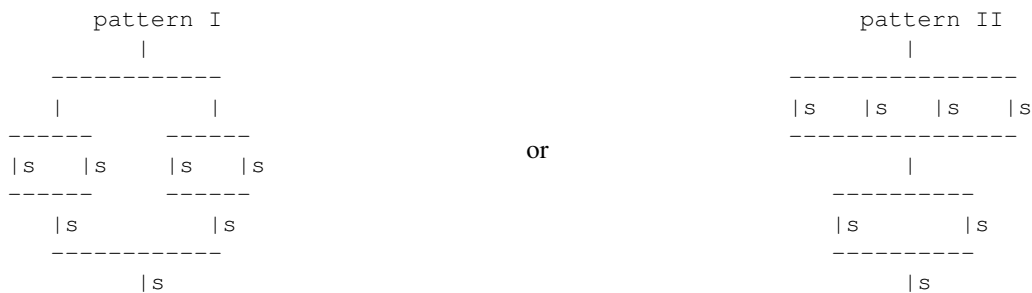
Consider the following SaC code:

```
int ssum (int[m] v)
{
    s = 0;
    for (i=0; i<m; i++)
        s += v[i];
    return s;
}

int ssum (int[m,r:shp] a)
{
    b = {[i] -> ssum (a[i])};
    return ssum (b);
}

int main()
{
    a = iota( 8);
    return ssum (reshape ([2,2,2], a));
}
```

**2.a)** Which of the two concurrency pattern below does this code exhibit? NB: the symbols  $s$  indicate sequential summations. 5 pt.



Solution:  
The solution above implements pattern I.

**2.b)** Re-implement the second instance of `ssum` so that it implements the other pattern! 10 pt.

Solution:

```
int ssum (int[r:shp,m] a)
{
    v = {iv -> ssum (a[iv]) | iv < shp};
    return ssum (v);
}
```

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

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 + arrays

In this assignment we work with the WHILE programming language. See Appendix A for one of the versions defined in the lecture.

For the evaluation of WHILE style programs, we introduce the following data types:

```
:: State ::= '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`. Access to the state is provided with the operations `write` and `read`:

```
write :: Var a → Eval () | TC a
write v a = E λs = (Result (), 'Data.Map'.put v (dynamic a) s)

read :: Var → Eval a | TC a
read v = E λs = case 'Data.Map'.get v s of
    ?Just (x :: a) = (Result x, s)
    ?Just _       = (Error ("read: var " <+ v <+ " has wrong type"), s)
    _              = (Error ("read: var " <+ v <+ " is undefined"), s)
```

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) We extend the WHILE DSL with support for arrays. The following language elements are added to expressions: 8 pt.

- `VarA v`: variable `v` refers to an array.
- `Gen low up`: generates an array with first element (at index 0) having the value from expression `low` and last element having the value from expression `up` (in case `low` exceeds `up`, an error result should be generated that an array cannot be empty).
- `Select i arr`: select element at index `i` from expression `i` in the array from expression `arr` (in case of an index out of bounds error, an error result should be generated).

and the statements are extended with an array element update:

- `Upd v i x`: replaces element at index from expression `i` in array variable `v` with the value from expression `x` (in case of an index out of bounds error, an error result should be generated).

Please note that for this assignment it is not required to use Clean arrays, Clean lists will do. Generating a Clean list with bound values `low` and `up` is done with `[low..up]`; selecting the element at index `j` from list `l` is done with `l !! j`, and updating an element at index `j` to a new value `x` in list `l` is done with `updateAt j x l`.

Fix the type `Expr a` given in Appendix A by adding the `BM` bi-maps needed for type-safe evaluation and extend it with cases for `VarA`, `Gen`, and `Select`. Extend the type `Stmt` with the case `Upd`.

Solution:

```
:: Expr a
= Lit a
| Add (BM a Int) (Expr Int) (Expr Int)
| Sub (BM a Int) (Expr Int) (Expr Int)
|  $\exists$ b: Leq (BM a Bool) (Expr b) (Expr b) & <, TC b
| Not (BM a Bool) (Expr Bool)
| Var (BM a Int) Var
| VarA (BM a [Int]) Var
| Gen (BM a [Int]) (Expr Int) (Expr Int)
| Select (Expr Int) (Expr [a])

:: Stmt
= $\exists$ a: ( $\Rightarrow$ ) infix 2 Var (Expr a) & TC a
| Upd Var (Expr Int) (Expr Int)
| (..) infixr 1 Stmt Stmt
| If (Expr Bool) Stmt Stmt
| While (Expr Bool) Stmt
| Skip
```

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

8 pt.

`evalE :: (Expr a) State → 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 >>= \vx = evalE y >>= \vy = pure (bm.ba (vx + vy))
    Sub bm x y  = evalE x >>= \vx = evalE y >>= \vy = pure (bm.ba (vx - vy))
    Leq bm x y  = evalE x >>= \vx = evalE y >>= \vy = pure (bm.ba (vx ≤ vy))
    Not bm x    = evalE x >>= \vx = pure (bm.ba (not vx))
    Var bm v    = read v >>= \vv = pure (bm.ba vv)
    VarA bm v    = read v >>= \va = pure (bm.ba va)
    Gen bm l u   = evalE l >>= \vl = evalE u >>= \vu = if (vl > vu)
                                                         (fail "empty array")
                                                         (pure (bm.ba [vl..vu]))
    Select i a   = evalE i >>= \vi = evalE a >>= \va = if (vi < 0 || vi ≥ length va)
                                                         (fail "index out of range")
                                                         (pure (va !! vi))
```

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

8 pt.

`evalS :: Stmt → Eval ()`

Solution:

```
evalS :: Stmt → Eval ()
evalS stmt = case stmt of
    v ← e      = evalE e >>= λx = write v x
    Upd v i e  = evalE i >>= λvi =
        evalE e >>= λve =
            read v >>= λva =
                if (vi < 0 || vi ≥ length va)
                    (fail "index out of range")
                (write v (updateAt vi ve va))
    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 ()
```

---

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 `Evaluator` to obtain a fresh variable name:

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

```
fresh :: Eval (Var` a)
```

```
fresh = E λs = (Result ("v" <+ 'Data.Map'.mapSize s), s)
```

An initial framework for the shallow and tagless WHILE DSL is given below. Finish all the missing components (...) 8 pt. of the framework.

```
class expr v
where lit      :: a → v a
      add      :: (v a) (v a) → v a | + a
      sub      :: (v a) (v a) → v a | - a
      leq      :: (v a) (v a) → v Bool | < a
      neg      :: (v Bool) → v Bool
      gen      :: ...    // the shallow, tagless version of Gen
      sel      :: ...    // the shallow, tagless version of Select

class vars v a
where var      :: (Var` a) → v a
      def      :: ...    // the shallow, tagless version of variable binding
      (.=) infixr 2 :: ... // the shallow, tagless version of =.

class stmt v
where (...) infixr 1 :: (v a) (v b) → v b
      upd`      :: ...    // the shallow, tagless version of Upd
      if`       :: (v Bool) (v a) (v a) → v a
      while`    :: (v Bool) (v a) → v ()
      skip`     :: v ()
```

```
:: In a b = In infix 0 a b
```

```
instance expr Eval
where lit x      = pure x
      add e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 + v2)
      sub e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 - v2)
      leq e1 e2 = e1 >>= λv1 = e2 >>= λv2 = pure (v1 ≤ v2)
```



```

        neg e      = e >>= \vb => pure (not vb)
        gen ...
        sel ...
instance vars Eval a | TC a
  where var      ...
        def      ...
        (.=) ...
instance stmt Eval
  where (... ) s t = s >>= \_ => t
        if' c t e = c >>= \b => if b t e
        while' c b = if' c (b ... while' c b) skip'
        skip'      = pure ()
        upd' ...

```

Solution:

```

class expr v
  where ...
    gen :: (v Int) (v Int) -> v [Int]
    sel :: (v Int) (v [Int]) -> v Int
class vars v a
  where ...
    def :: (Var' a) -> In a (v b) -> v b
    (.=) infixr 2 :: (Var' a) (v a) -> v a
class stmt v
  where upd' :: (v [Int]) (v Int) (v Int) -> v ()
        ...
instance expr Eval
  where ...
    gen el eu    = el >>= \v1 => eu >>= \vu => pure [v1..vu]
    sel ei ea    = ei >>= \vi => ea >>= \va => pure (va !! vi)
instance vars Eval a | TC a
  where var v      = read v
        def f      = fresh >>= \v => let (a In e) = f v in write v a >> | e
        (.=) v e    = e >>= \ve => write v ve >>= \_ => pure ve
instance stmt Eval
  where upd' arr i x = arr >>= \va => i >>= \vi => x >>= \vx => pure (updateAt vi vx va) >>= \_ => pure ()
        ...

```

---

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

---

## 5 Task Oriented Programming

In this assignment we create a couple of tasks to allow people to chat with each other. For the chat, a shared data source is globally accessible:

```
chatsSDS :: SimpleSDSLens [Message]
chatsSDS = sharedStore "chats" []
```

A message contains information about when the message is created (roughly), the sender, the receivers, and message text:

```
:: Message
= { when      :: DateTime    // predefined in iTask
  , sender    :: User        // predefined in iTask
  , receivers :: [User]      // non-empty list of recipients
  , message   :: String      // non-empty text
}
```

**derive class** iTask Message

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:

3 pt.

```
returnIf :: (Task a) (a → Bool) → Task a | iTask a
```

(`returnIf t p`) evaluates `t`, and as soon as `t` has a (stable or unstable) task value (`TaskValue x _`) for which predicate (`p x`) evaluates to `True`, then (`returnIf t p`) returns `x`.

Solution:

```
returnIf :: (Task a) (a → Bool) → Task a | iTask a
returnIf t cond = t >>* [OnAction (Action "Ok") (ifValue cond return)]
```

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

3 pt.

```
enterMessage :: [User] → Task Message
```

(`enterMessage users`) lets the current user (available from the SDS `currentUser`) generate a new message that contains the current date and time (available from the SDS `currentDateTime`), a *non-empty* subset of users, and a *non-empty* message text. If these conditions hold, then the created message is returned as a stable task value.

Solution:

```
enterMessage :: [User] → Task Message
enterMessage users
  = get currentUser >>- λme =
    returnIf (Hint "Please select at least one receiver" @>>
      enterMultipleChoice [] users) (not o isEmpty) >>- λreceivers =
    returnIf (Hint "Please enter a non-empty message" @>>
      updateInformation [] "Type your message here") (≠ "") >>- λmessage =
    get currentDateTime >>- \now =
    return {Message | when = now, sender = me, receivers = receivers, message = message}
```

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

3 pt.

```
addMessage :: [User] (SimpleSDSLens [Message]) → Task [Message]
```

(`addMessage users sds`) lets the current user create a new message (with (`enterMessage users`)) and adds that message with an *atomic update* to the argument SDS `sds`.

**Solution:**

```
addMessage :: [User] (SimpleSDSLens [Message]) → Task [Message]
addMessage users msgs
  = enterMessage users >>- λmsg = upd (λchat = chat ++ [msg]) msgs
```

**5.d) Implement the task function:**

4 pt.

```
session :: [User] (SimpleSDSLens [Message]) → Task ()
```

(session users sds) lets every user in users perform the task that views the current content of the given SDS sds, and offers them two actions:

1. *New message*: at all times it is possible for the user to create a new message and add it to sds. This can be repeated arbitrarily many times.
2. *Stop*: at all times it is possible for the user to stop participating in the session.

**Solution:**

```
session :: [User] (SimpleSDSLens [Message]) → Task ()
session users chats
  = allTasks [who @: chat users chats \\ who ← users] @! ()

chat :: [User] (SimpleSDSLens [Message]) → Task [Message]
chat users msgs
  = viewSharedInformation [ViewAs (map toString)] msgs
  >>* [ OnAction (Action "New message") (always (addMessage users msgs >>- λ_ = chat users msgs))
      , OnAction (Action "Stop") (hasValue return)
      ]
// not really needed, makes output more compact:
instance toString Message
  where toString {when, sender, receivers, message}
    = sender <+ " ": " " <+ message
```

---

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

---

## A While

In this appendix you find the deeply embedded representation of the WHILE language *without* the  $\text{BM}$  bi-maps.

```

:: Expr a
= Lit a
| Var      Var
| Plus     (Expr Int) (Expr Int)
| Not      (Expr Bool)
| And      (Expr Bool) (Expr Bool)
|  $\exists$ b:Leq (Expr b) (Expr b) & <, TC b

:: Stmt
= ( $\Leftarrow$ ) infix 2 Var (Expr Int)
| ( $\Leftarrow$ ) infixr 1 Stmt Stmt
| If      (Expr Bool) Stmt Stmt
| While   (Expr Bool) Stmt
| Skip

:: BM a b = {ab :: a  $\rightarrow$  b, ba :: b  $\rightarrow$  a}
bm = {ab = id, ba = id}

```

A program that creates an array and reverses it looks similar to this:

```

reverse :: Int Int  $\rightarrow$  Stmt
reverse low up
  = "low" =. Lit low ::.
    "up" =. Lit up ::.
    "data" =. Gen (Var "low") (Var "up") ::.           // data = [low..up]
    "i" =. Sub (Var "up") (Var "low") ::.              // i = up - low
    "j" =. Lit 0 ::.                                    // j = 0
    While (Leq (Var "j") (Var "i")) (                 // while (j  $\leq$  i) {
      "y" =. Select (Var "i") (VarA "data") ::.        //   y = data[i]
      Upd "data" (Var "i") (Select (Var "j") (VarA "data")) ::. //   data[i] = data[j]
      Upd "data" (Var "j") (Var "y") ::.              //   data[j] = y
      "i" =. Sub (Var "i") (Lit 1) ::.                 //   i = i - 1
      "j" =. Add (Var "j") (Lit 1)                     //   j = j + 1
    )                                                  // }

```

## B Clean

```

// Monad and friends. Import these definitions as
import Control.Monad, Control.Monad.Fail, Control.Applicative, Data.Functor
class pure :: a  $\rightarrow$  f a
class Functor f
where fmap :: (a  $\rightarrow$  b) (f a)  $\rightarrow$  f b
      (<$>) infixl 4 :: (a  $\rightarrow$  b) (f a)  $\rightarrow$  f b | Functor f
      (<$>) f fa := fmap f fa
class (<*>) infixl 4 :: (f a  $\rightarrow$  b) (f a)  $\rightarrow$  f b
class Monad m | Applicative m
where bind :: !(m a) (a  $\rightarrow$  m b)  $\rightarrow$  m b
      (>>=) infixl 1 :: (m a) (a  $\rightarrow$  m b)  $\rightarrow$  m b | Monad m
      (>>=) ma a2mb := bind ma a2mb
      (>>) infixl 1 :: (m a) (m b)  $\rightarrow$  m b | Monad m
      (>>) ma mb := ma >>= ( $\lambda$ _  $\rightarrow$  mb)
class MonadFail m | Monad m
where fail :: String  $\rightarrow$  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

```

## C 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

```