Translation from ML AST to JS AST

```
c ::=
    unit            null
    bool            bool
    int             number
    float           number
    char
    string          string
    bytes

[e_ml] x_js s_js   //x_js -- where we save result of ml-expression translation
                   //s_js -- what we should do next
e ::=
    const           [C] x s = let x = C; s
    var             [x] y s = let y = x; s
    name            [name] x s = let x = name; s
    let             [let x = e1 in e2] y s = [e1] x ([e2] y s)
    app             [f x] y s = let y = f [x]; s
    fun             [fun x => e] f s = let f = (x) => {[e] _res (return _res)}
    match           [match e with |p_i -> e_i] x s =
                        [e] _match_e ([p_1] _match_e ([e_1] x None)
                                      ([p_2] _match_e ([e_2] x None)
                                                 ...
                                                 ([p_n] _match_e ([e_n] x None) Exp))); s
    coerce          [(e, from_t, to_t)] x s = [e] x:to_t s
    CTor            [C p1 .. pn] x s = let x = {_tag: "C", _1: [p1], .. , _n: [pn]}; s
    Seq             [e1; ..;en] x s = [e1] _ ([e2] _ (.. ([en] x s)))
    Tuple           [(e1, .., en)] x s = let x = [[e1], .., [en]]; s
    Record          [g1:e1, .., gn:en] x s  = let x = {_tag: "Record",
                                                      _g1: [e1], .., _gn: [en]}; s
    Proj            [(e, name)] x s = let x = [e]._name; s
    If              [if e then e1 else e2] x s =
                        [e] _cond (if(_cond){[e1] x None} else {[e2] x None}); s
    Raise
    Try
```

```
[p_ml] e_js s1_then s2_else
p ::=
    wild            [_] e s1 s2 = s1
    const           [C] e s1 s2 = if (e == C) s1 else s2
    var             [x] e s1 s2 = let x = e; s1
    CTor            [C p1 .. pn] e s1 s2 =
                        if(e._tag === "C")
                            [p1] e._1 ([p2] e._2 (.. ([pn] e._n s1 s2)) s2) s2
                        else s2
                    //w/o repeating s2:
                    [C p] e s1 s2 =
                        { let _valid = true
                          if(e._tag === "C")
                            [p1] e._1
                                ([p2] e._2
                                    (.. ([pn] e._n s1 (_valid = false)))
                                _valid = false)
                            _valid = false
                          else  _valid = false
                          if (!_valid) s2 }
    branch          [p1|p2|..|pn] e s1 s2 =
                        [p1] e s1 ([p2] e s1 (.. ([pn] e s1 s2)))
    record          [g1:p1,.., gn:pn] e s1 s2 =
                        [p1] e._g1 ([p2] e._g2 (.. ([pn] e._gn s1 s2)) s2) s2
    tuple           [(p1,.., pn)] e s1 s2 =
                        [p1] e[1] ([p2] e[2] (.. ([pn] e[n] s1 s2)) s2) s2
[p when g] e s1 s2 = [p] e ([g] _x (if (_x) s1 else s2)) s2
//w/o repeating s2:
[p when g] e s1 s2 =
    { let _valid = true
      [p] e ([g] _x (if(_x) s1 else _valid = false)) (_valid = false)
      if (!_valid) s2 }
```

Cases, where we can avoid repeat "_valid = false", i.e.

```
[p] e s1 s2 = if (be_1[e] && be_2[e] && ..)
              {let fv_1 = ..
               let fv_2 = ..
               s1} else s2
```

```
sp ::=                          [sp] e s1 s2 = if (be) {s; s1}
    var x                                      else s2
    const C                     [sp] e ==> be, s
    CTop C x
    Tuple (x, y)

[sp when e] l s1 s2 = { let _valid = true
                        if (be[l]){s; [e] _valid (if (_valid) s1)}
                        else _valid = false
                        if (!_valid) s2 }
[x] e ==> true, let x = e
[C] e ==> (C == e), None
[C sp] e ==> (e._tag === "C" && be), s
             where [sp] e.value ==> be, s
[(sp_1, .., sp_n)] e ==> (be_1 && .. && be_n); s_1; ..; s_n
                          [sp_1] e._f1 ==> be_1, s_1
                          ...
                          [sp_n] e._fn ==> be_n, s_n
```

Types

```
t ::=
    int                         [int] = number
    bool                        [bool] = bool
    string                      [string] = string
    t1*t2*..*tn                 [(t1*t2*..*tn)]  = [[t1], .., [tn]]
    C t1 .. tn                  [C t1 .. tn] = C<[t1], .., [tn]>

C ::=
    type C x1 .. xn =           [..] = type C<x1 .. xn> = {_tag: "Record",
      {f1:t1, .., fn:tn}                                   _f1: [t1], .., _fn: [tn]}
    type C x1 .. xn  = t        [..] = type C<x1 .. xn> = [t]
    type C x1 .. xn  =          [..] = type C1<x1 .. xn> = {_tag: "C1", _0: [t1]}
        | C1 of t1                       ...
          ...                           type Cn<x1 .. xn> = {_tag: "Cn", _0: [tn]}
        | Cn of tn                      type C<x1 .. xn> = C1<x1 .. xn>| .. |Cn<x1 .. xn>
    type C x1 .. xn =           [..] = type C<x1 .. xn> = {_tag: "C",
        | C of t1 -> t2 -> tn                            _1: [t1], .., _n: [tn]}
```