Languages-beta: OC-L-07-Expressions *

The PLanCompS Project

`OC-L-07-Expressions.cbs` | PLAIN | PRETTY

OUTLINE

Language "`OCaml Light`"

*Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

7 Expressions

```
Syntax   E : expr  ::=  value-path
                     |  constant
                     |  ‘(’ expr ‘)’
                     |  ‘begin’ expr ‘end’
                     |  ‘(’ expr ‘:’ typexpr ‘)’
                     |  expr comma-expr+
                     |  expr ‘::’ expr
                     |  ‘[’ expr semic-expr* ‘]’
                     |  ‘[’ expr semic-expr* ‘;’ ‘]’
                     |  ‘[|’ expr semic-expr* ‘|]’
                     |  ‘[|’ expr semic-expr* ‘;’ ‘|]’
                     |  ‘{’ field ‘=’ expr semic-field-expr* ‘}’
                     |  ‘{’ field ‘=’ expr semic-field-expr* ‘;’ ‘}’
                     |  ‘{’ expr ‘with’ field ‘=’ expr semic-field-expr* ‘}’
                     |  ‘{’ expr ‘with’ field ‘=’ expr semic-field-expr* ‘;’ ‘}’
                     |  expr argument+
                     |  prefix-symbol expr
                     |  ‘-’ expr
                     |  ‘-.’ expr
                     |  expr infix-op-1 expr
                     |  expr infix-op-2 expr
                     |  expr infix-op-3 expr
                     |  expr infix-op-4 expr
                     |  expr infix-op-5 expr
                     |  expr infix-op-6 expr
                     |  expr infix-op-7 expr
                     |  expr infix-op-8 expr
                     |  expr ‘.’ field
                     |  expr ‘.(’ expr ‘)’
                     |  expr ‘.(’ expr ‘)’ ‘<-’ expr
                     |  ‘if’ expr ‘then’ expr (‘else’ expr)?
                     |  ‘while’ expr ‘do’ expr ‘done’
                     |  ‘for’ value-name ‘=’ expr (‘to’ | ‘downto’) expr ‘do’ expr ‘done’
                     |  expr ‘;’ expr
                     |  ‘match’ expr ‘with’ pattern-matching
                     |  ‘function’ pattern-matching
                     |  ‘fun’ pattern+ ‘->’ expr
                     |  ‘try’ expr ‘with’ pattern-matching
                     |  let-definition ‘in’ expr
                     |  ‘assert’ expr
        A : argument  ::=  expr
PM : pattern-matching  ::=  pattern ‘->’ expr pattern-expr*
                        |  ‘|’ pattern ‘->’ expr pattern-expr*
  LD : let-definition  ::=  ‘let’ (‘rec’)? let-binding and-let-binding*
     LB : let-binding  ::=  pattern ‘=’ expr
                        |  value-name pattern+ ‘=’ expr
                        |  value-name ‘:’ poly-typexpr ‘=’ expr
ALB : and-let-binding  ::=  ‘and’ let-binding
    CE : comma-expr  ::=  ‘,’ expr
    SE : semic-expr  ::=  ‘;’ expr
```

Rule ⟦ ‘(’ E ‘)’ ⟧ : expr = ⟦ E ⟧

Rule ⟦ ‘begin’ E ‘end’ ⟧ : expr = ⟦ E ⟧

Rule ⟦ ‘(’ E ‘:’ T ‘)’ ⟧ : expr = ⟦ E ⟧

Rule ⟦ E_1 E_2 A A^* ⟧ : expr = ⟦ (‘(’ E_1 E_2 ‘)’) A A^* ⟧

Rule ⟦ PS E ⟧ : expr = ⟦ (‘(’ PS ‘)’) E ⟧

Rule ⟦ ‘-’ E ⟧ : expr = ⟦ (‘(’ ‘~-’ ‘)’) E ⟧

Rule ⟦ ‘-.’ E ⟧ : expr = ⟦ (‘(’ ‘~-.’ ‘)’) E ⟧

Rule ⟦ E_1 *IO-1* E_2 ⟧ : expr = ⟦ (‘(’ *IO-1* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 *IO-2* E_2 ⟧ : expr = ⟦ (‘(’ *IO-2* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 *IO-3* E_2 ⟧ : expr = ⟦ (‘(’ *IO-3* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 *IO-4* E_2 ⟧ : expr = ⟦ (‘(’ *IO-4* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 *IO-5* E_2 ⟧ : expr = ⟦ (‘(’ *IO-5* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 ‘&’ E_2 ⟧ : expr = ⟦ E_1 ‘&&’ E_2 ⟧

Rule ⟦ E_1 ‘or’ E_2 ⟧ : expr = ⟦ E_1 ‘||’ E_2 ⟧

Rule ⟦ E_1 *IO-8* E_2 ⟧ : expr = ⟦ (‘(’ *IO-8* ‘)’) E_1 E_2 ⟧

Rule ⟦ E_1 ‘.(’ E_2 ‘)’ ⟧ : expr = ⟦ ‘array_get’ E_1 E_2 ⟧

Rule ⟦ E_1 ‘.(’ E_2 ‘)’ ‘<-’ E_3 ⟧ : expr = ⟦ ‘array_set’ E_1 E_2 E_3 ⟧

Rule ⟦ ‘if’ E_1 ‘then’ E_2 ⟧ : expr = ⟦ ‘if’ E_1 ‘then’ E_2 ‘else’ (‘(’ ‘)’) ⟧

Rule ⟦ ‘fun’ P ‘->’ E ⟧ : expr = ⟦ ‘function’ P ‘->’ E ⟧

Rule ⟦ ‘fun’ P P^+ ‘->’ E ⟧ : expr = ⟦ ‘fun’ P ‘->’ (‘fun’ P^+ ‘->’ E) ⟧

Rule ⟦ ‘[’ E SE^* ‘;’ ‘]’ ⟧ : expr = ⟦ ‘[’ E SE^* ‘]’ ⟧

Rule ⟦ ‘[|’ E SE^* ‘;’ ‘|]’ ⟧ : expr = ⟦ ‘[|’ E SE^* ‘|]’ ⟧

Rule ⟦ ‘{’ F ‘=’ E SFE^* ‘;’ ‘}’ ⟧ : expr = ⟦ ‘{’ F ‘=’ E SFE^* ‘}’ ⟧

Rule ⟦ ‘{’ E_1 ‘with’ F ‘=’ E_2 SFE^* ‘;’ ‘}’ ⟧ : expr =
⟦ ‘{’ E_1 ‘with’ F ‘=’ E_2 SFE^* ‘}’ ⟧

Rule ⟦ ‘|’ P ‘->’ E PE^* ⟧ : pattern-matching = ⟦ P ‘->’ E PE^* ⟧

Rule ⟦ VN ‘:’ PT ‘=’ E ⟧ : let-binding = ⟦ VN ‘=’ E ⟧

Rule ⟦ VN P^+ ‘=’ E ⟧ : let-binding = ⟦ VN ‘=’ (‘fun’ P^+ ‘->’ E) ⟧

```
Semantics  evaluate⟦ _ : expr ⟧ : ⇒ implemented-values
     Rule  evaluate⟦ VP ⟧ = bound(value-name⟦ VP ⟧)
     Rule  evaluate⟦ CNST ⟧ = value⟦ CNST ⟧
     Rule  evaluate⟦ ‘(’ E ‘:’ T ‘)’ ⟧ = evaluate⟦ E ⟧
     Rule  evaluate⟦ E_1 ‘,’ E_2 CE* ⟧ =
             tuple(evaluate-comma-sequence⟦ E_1 ‘,’ E_2 CE* ⟧)
     Rule  evaluate⟦ E_1 ‘::’ E_2 ⟧ = cons(evaluate⟦ E_1 ⟧, evaluate⟦ E_2 ⟧)
     Rule  evaluate⟦ ‘[’ E SE* ‘]’ ⟧ = [evaluate-semic-sequence⟦ E SE* ⟧]
     Rule  evaluate⟦ ‘[|’ E SE* ‘|]’ ⟧ =
             vector(
               left-to-right-map(
                 allocate-initialised-variable(implemented-values, given),
                 evaluate-semic-sequence⟦ E SE* ⟧))
     Rule  evaluate⟦ ‘[|’ ‘|]’ ⟧ = vector( )
     Rule  evaluate⟦ ‘{’ F ‘=’ E SFE* ‘}’ ⟧ =
             record(collateral(evaluate-field-sequence⟦ F ‘=’ E SFE* ⟧))
     Rule  evaluate⟦ ‘{’ E_1 ‘with’ F ‘=’ E_2 SFE* ‘}’ ⟧ =
             record(
               map-override(
                 evaluate-field-sequence⟦ F ‘=’ E_2 SFE* ⟧,
                 checked record-map(evaluate⟦ E_1 ⟧)))
     Rule  evaluate⟦ CSTR E ⟧ =
             variant(constr-name⟦ CSTR ⟧, evaluate⟦ E ⟧)
Otherwise  evaluate⟦ E_1 E_2 ⟧ =
             apply(evaluate⟦ E_1 ⟧, evaluate⟦ E_2 ⟧)
     Rule  evaluate⟦ E ‘.’ F ⟧ =
             record-select(evaluate⟦ E ⟧, field-name⟦ F ⟧)
     Rule  evaluate⟦ E_1 ‘&&’ E_2 ⟧ =
             if-true-else(evaluate⟦ E_1 ⟧, evaluate⟦ E_2 ⟧, false)
     Rule  evaluate⟦ E_1 ‘||’ E_2 ⟧ =
             if-true-else(evaluate⟦ E_1 ⟧, true, evaluate⟦ E_2 ⟧)
     Rule  evaluate⟦ ‘if’ E_1 ‘then’ E_2 ‘else’ E_3 ⟧ =
             if-true-else(evaluate⟦ E_1 ⟧, evaluate⟦ E_2 ⟧, evaluate⟦ E_3 ⟧)
     Rule  evaluate⟦ ‘while’ E_1 ‘do’ E_2 ‘done’ ⟧ =
             while(evaluate⟦ E_1 ⟧, effect(evaluate⟦ E_2 ⟧))
     Rule  evaluate⟦ ‘for’ VN ‘=’ E_1 ‘to’ E_2 ‘do’ E_3 ‘done’ ⟧ =
             effect(
               left-to-right-map(
                 case-match(pattern-bind(value-name⟦ VN ⟧), evaluate⟦ E_3 ⟧),
                 integer-sequence(evaluate⟦ E_1 ⟧, evaluate⟦ E_2 ⟧)))
     Rule  evaluate⟦ ‘for’ VN ‘=’ E_1 ‘downto’ E_2 ‘do’ E_3 ‘done’ ⟧ =
             effect(
               left-to-right-map(
                 case-match(pattern-bind(value-name⟦ VN ⟧), evaluate⟦ E_3 ⟧),
                 reverse integer-sequence(evaluate⟦ E_2 ⟧, evaluate⟦ E_1 ⟧)))
     Rule  evaluate⟦ E_1 ‘;’ E_2 ⟧ =
             sequential(effect(evaluate⟦ E_1 ⟧), evaluate⟦ E_2 ⟧)
     Rule  evaluate⟦ ‘match’ E ‘with’ PM ⟧ =
             give(
               evaluate⟦ E ⟧,
               else(match⟦ PM ⟧, throw(ocaml-light-match-failure)))
     Rule  evaluate⟦ ‘function’ PM ⟧ =
             function closure(
```

Expression sequences and maps

Semantics evaluate-comma-sequence⟦ _ : (expr comma-expr*) ⟧ : (⇒ implemented-values)$^+$

Rule evaluate-comma-sequence⟦ E_1 ‘,’ E_2 CE^* ⟧ =

evaluate⟦ E_1 ⟧, evaluate-comma-sequence⟦ E_2 CE^* ⟧

Rule evaluate-comma-sequence⟦ E ⟧ = evaluate⟦ E ⟧

Semantics evaluate-semic-sequence⟦ _ : (expr semic-expr*) ⟧ : (⇒ implemented-values)$^+$

Rule evaluate-semic-sequence⟦ E_1 ‘;’ E_2 SE^* ⟧ =

evaluate⟦ E_1 ⟧, evaluate-semic-sequence⟦ E_2 SE^* ⟧

Rule evaluate-semic-sequence⟦ E ⟧ = evaluate⟦ E ⟧

Semantics evaluate-field-sequence⟦ _ : (field ‘=’ expr semic-field-expr*) ⟧ : (⇒ envs)$^+$

Rule evaluate-field-sequence⟦ F_1 ‘=’ E_1 ‘;’ F_2 ‘=’ E_2 SFE^* ⟧ =

{field-name⟦ F_1 ⟧ ↦ evaluate⟦ E_1 ⟧},

evaluate-field-sequence⟦ F_2 ‘=’ E_2 SFE^* ⟧

Rule evaluate-field-sequence⟦ F ‘=’ E ⟧ = {field-name⟦ F ⟧ ↦ evaluate⟦ E ⟧}

Matching

Semantics match⟦ _ : pattern-matching ⟧ : (implemented-values ⇒ implemented-values)$^+$

Rule match⟦ P_1 ‘->’ E_1 ‘|’ P_2 ‘->’ E_2 PE^* ⟧ =

match⟦ P_1 ‘->’ E_1 ⟧, match⟦ P_2 ‘->’ E_2 PE^* ⟧

Rule match⟦ P ‘->’ E ⟧ = case-match(evaluate-pattern⟦ P ⟧, evaluate⟦ E ⟧)

Value definitions

Semantics define-values⟦ _ : let-definition ⟧ : ⇒ environments

Rule define-values⟦ ‘`let`’ LB ALB^* ⟧ = define-values-nonrec⟦ LB ALB^* ⟧

Rule define-values⟦ ‘`let rec`’ LB ALB^* ⟧ =

recursive(

set(bound-ids-sequence⟦ LB ALB^* ⟧),

define-values-nonrec⟦ LB ALB^* ⟧)

Semantics define-values-nonrec⟦ _ : (let-binding and-let-binding*) ⟧ : ⇒ environments

Rule define-values-nonrec⟦ LB_1 ‘`and`’ LB_2 ALB^* ⟧ =

collateral(define-values-nonrec⟦ LB_1 ⟧, define-values-nonrec⟦ LB_2 ALB^* ⟧)

Rule define-values-nonrec⟦ P ‘=’ E ⟧ =

else(

match(evaluate⟦ E ⟧, evaluate-pattern⟦ P ⟧),

throw(ocaml-light-match-failure))

Semantics bound-ids-sequence⟦ _ : (let-binding and-let-binding*) ⟧ : ids$^+$

Rule bound-ids-sequence⟦ LB ⟧ = bound-id⟦ LB ⟧

Rule bound-ids-sequence⟦ LB_1 ‘`and`’ LB_2 ALB^* ⟧ =

bound-id⟦ LB_1 ⟧, bound-ids-sequence⟦ LB_2 ALB^* ⟧

Semantics bound-id⟦ _ : let-binding ⟧ : ids

Rule bound-id⟦ VN ‘=’ E ⟧ = value-name⟦ VN ⟧

Otherwise bound-id⟦ LB ⟧ = fail