# Denotational Semantics of MicroHaskell

## Abstract Syntax:

| **Syntax Domains:** | **Abstract Syntax Rules:** |
|---|---|
| **PG**: Program | **PG ::= E** |
| **E**: Expression | **E ::= let D in E \| if $E_1$ then $E_2$ else $E_3$ \|** |
| | **N \| Id \| Id A \|** |
| | **$E_1$ + $E_2$ \| $E_1$ - $E_2$ \| $E_1$ * $E_2$ \| $E_1$ / $E_2$** |
| | **$E_1$ == $E_2$ \| $E_1$ /= $E_2$ \| $E_1$ < $E_2$ \| $E_1$ <= $E_2$ \| $E_1$ > $E_2$ \| $E_1$ >= $E_2$ \|** |
| | **$E_1$ && $E_2$ \| $E_1$ \|\| $E_2$ \|** |
| | **$E_1$ : $E_2$ \| head E \| tail E \| [] \|** |
| **D**: Declaration | **D ::= $D_1$ $D_2$ \| Id = E \| Id P = E** |
| **P**: Parameter | **P ::= Id \| Id P** |
| **A**: Argument | **A ::= E \| E A** |
| **N**: Integer | |
| **Id**: Identifier | |

## Semantic Domains:

I. Boolean Values

Domain b $\in$ Boolean = { *true, false* }$^\circ$

II. Integers

Domain n $\in$ N = {. . . , -2, -1, 0, 1, 2, . . .}$^\circ$

Operations:

$AO \llbracket \mathbf{+} \rrbracket\ n_1\ n_2 = (n_1 + n_2)$

$AO \llbracket \mathbf{-} \rrbracket\ n_1\ n_2 = (n_1 - n_2)$

$AO \llbracket \mathbf{*} \rrbracket\ n_1\ n_2 = (n_1 \times n_2)$ — *AO* returns an integer value

$AO \llbracket \mathbf{/} \rrbracket\ n_1\ n_2 = \mathit{if}\ (n_2 = 0)\ \mathit{then}\ \top\ \mathit{else}\ (n_1 / n_2)$

$RO \llbracket \mathbf{==} \rrbracket\ n_1\ n_2 = (n_1 = n_2)$

$RO \llbracket \mathbf{/=} \rrbracket\ n_1\ n_2 = (n_1 \neq n_2)$

$RO \llbracket \mathbf{<} \rrbracket\ n_1\ n_2 = (n_1 < n_2)$

$RO \llbracket \mathbf{<=} \rrbracket\ n_1\ n_2 = (n_1 \leq n_2)$

$RO \llbracket \mathbf{>} \rrbracket\ n_1\ n_2 = (n_1 > n_2)$

$RO \llbracket \mathbf{>=} \rrbracket\ n_1\ n_2 = (n_1 \geq n_2)$ — *RO* returns a boolean value

III. Lists

Domain $\iota \in$ List = $N^*$

Operations:

*nil* : List

*cons* : N $\rightarrow$ List $\rightarrow$ List

*head* : List $\rightarrow$ N

*tail* : List $\rightarrow$ List

*eqlist* : List $\times$ List $\rightarrow$ Boolean

*neqlist* : List $\times$ List $\rightarrow$ Boolean

IV. Identifiers

Domain id $\in$ Identifier

V. Types

Domain t $\in$ Type = { *integer*, *list*, *boolean* }

VI. Categories

Domain c $\in$ Category = { *variable, function* }

VII. Values

Domain v $\in$ Value = N $\cup$ List $\cup$ B $\cup$ FuncDenot

VIII. Expressible-values

Domain e $\in$ Expressible-value = $\{(\text{Type} \times \text{Value})\}^\circ$

Operations:

*expr-value* : Type $\rightarrow$ Value $\rightarrow$ Denotable-value

*expr-value* t v = (t, v) "**constructs** the expressible value tuple"

*type* : Expressible-value $\rightarrow$ Type

*type* e = (e $\downarrow$ 1) "**selects** the type component"

*value* : Expressible-value $\rightarrow$ Value

*value* e = (e $\downarrow$ 2) "**selects** the value component"

IX. Denotable-values

Domain d $\in$ Denotable-value = $\{(\text{Category} \times \text{Type} \times \text{Value})\}^\circ$

Operations:

*denotable-value* : Category $\rightarrow$ Type $\rightarrow$ Value $\rightarrow$ Denotable-value

*denotable-value* c t v = (c, t, v) "**constructs** denotable value tuple"

*category* : Denotable-value $\rightarrow$ Category

*category* d = (d $\downarrow$ 1) "**selects** the category component"

*type* : Denotable-value $\rightarrow$ Type

*type* d = (d $\downarrow$ 2) "**selects** the type component"

*value* : Denotable-value $\rightarrow$ Value

*value* d = (d $\downarrow$ 3) "**selects** the value component"

*expr-value* : Denotable-value → Expressible-value

*expr-value* d = ((d ↓ 2), (d ↓ 3))  "**selects** expressible value part"

IX. Functions

Domain f ∈ FuncDenot = (Par* × (Environment → Expressible-value) × Environment)

/* Sequence of Formal Parameters, Function Body, Function definition Environment */

Operations:

*funcDenot* : Formal Parameter → Function Body → Environment → FuncDenot

*funcDenot* fp fbody $env_{def}$ = (fp, fbody, $env_{def}$)

"**constructs** FuncDenot tuple"

*formpar* : FuncDenot → Par*

*formpar* f = (f ↓ 1)  "**selects** formal parameter component"

*funcbody* : FuncDenot → (Environment → Expressible-value)

*funcbody* f = (f ↓ 2)  "**selects** function body component"

*funcEnv* : FuncDenot → Environment

*funcEnv* f = (f ↓ 3)  "**selects** function defining environment"

X. Environment

Domain env ∈ Environment = Id → Denotable-value

Operations:

*init-env* : Id → Denotable-value

*init-env* = λ Id . ⊥

*access-env* : Id → Environment → Denotable-value

*access-env* Id env = env⟦ Id ⟧

*update-env* : Id → Denotable-value → Environment → Environment

*update-env* Id d env = env[d / Id ]

## Semantic Functions:

*M* : Program → Expressible-value

*D* : Declaration → Environment → Environment

*E* : Expression → Environment → Expressible-value

*PM* : Actual Parameter → Environment → Formal Parameter → Environment → Environment

## Semantic Equations:

*M* ⟦**E**⟧ = *E* ⟦**E**⟧ (*init-env*)

*D* ⟦$\mathbf{D_1}$ $\mathbf{D_2}$⟧ env = *D* ⟦$\mathbf{D_2}$⟧ (*D* ⟦$\mathbf{D_1}$⟧ env)

*D* ⟦**Id = E**⟧ env =
```
let e = (E ⟦E⟧ env)                                  /* Variable Definition */
in    let d = (denotable-value 'variable' (type e) (value e))
      in    (update-env ⟦Id⟧ d env)
end   end
```

```
D ⟦Id P = E⟧ env_d =                                          /* Function Definition */
        let d = (denotable-value 'function' 'undefined' f )
        in      let env_d' = (update-env ⟦Id⟧ d env_d)
                in env_d'
                end     where f = (funcDenot ⟦P⟧ E⟦E⟧ env_d' )     /* constructing funcDenot */
        end

E ⟦let D in E⟧ env =  E ⟦E⟧ (D ⟦D⟧ env)

E ⟦Id⟧ env =  let d = (access-env ⟦Id⟧ env )                  /* Variable Access */
              in      if (category d = function) then ⊤
                      else (expr-val d)
              end

E ⟦Id A⟧ env_call =     let d = (access-env ⟦Id⟧ env_call )          /* Function Call */
                        in      if (category d ≠ function)  then ⊤
                                else    let env_def' = (PM ⟦A⟧ env_call (formpar f)  (funcEnv f))
                                        in      (funcbody f) env_def'        where f = (value d)
                        end             end

PM ⟦E⟧ env_call ⟦Id⟧ env_def =
        let e_actual = (E ⟦E⟧ env_call)
        in      let d = (denotable-value 'variable' (type e_actual) (value e_actual))
                in      (update-env ⟦Id⟧ d env_def )            "Parameter matching"
        end     end

PM ⟦E A⟧  env_call ⟦Id P⟧  env_def  =  let env_def1 = PM ⟦E⟧ env_call ⟦Id⟧ env_def
                                       in      PM ⟦A⟧ env_call ⟦P⟧ env_def1
                                       end     "Parameter matching - more than one parameter"

PM ⟦E A⟧ env_call ⟦Id⟧ env_def = ⊤          "Parameter mismatch"
PM ⟦E⟧ env_call ⟦Id P⟧ env_def = ⊤          "Parameter mismatch"

E ⟦if E1 then E2 else E3⟧ env =
              let e_1 = E ⟦E1⟧ env
              in      if (type e_1 ≠ boolean)  then  ⊤
                      else    let e_2 = E ⟦E2⟧ env and  e_3 = E ⟦E3⟧ env
                              in      if (type e_2 ≠ type e_3 = true)  then  ⊤
                                      else    if (value e_1 = true) then e_2
                                              else e_3
                              end
              end
```

$E$ ⟦**E₁ : E₂**⟧ env = let $e_1 = E$ ⟦**E₁**⟧ env and $e_2 = E$ ⟦**E₂**⟧ env
in if ((*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *list*)) then $\top$
else (*expr-val list* (*cons* (*value* $e_1$) (*value* $e_2$)))
end

$E$ ⟦ [] ⟧ env = (*expr-val list nil*)

$E$ ⟦**tail E**⟧ env = let $e_1 = E$ ⟦**E**⟧ env
in if (*type* $e_1 \neq$ *list*) then $\top$
else (*expr-val list* (*tail* (*value* $e_1$)))
end

$E$ ⟦**head E**⟧ env = let $e_1 = E$ ⟦**E**⟧ env
in if (*type* $e_1 \neq$ *list*) then $\top$
else (*expr-val integer* (*head* (*value* $e_1$)))
end

$E$ ⟦**E₁ aop E₂**⟧ env = let $e_1 = E$ ⟦**E₁**⟧ env and $e_2 = E$ ⟦**E₂**⟧ env
in if ((*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *integer*)) then $\top$
else (*expr-val integer* (*AO* ⟦**aop**⟧ (*value* $e_1$) (*value* $e_2$)))
end where **aop** $\in$ { **+** , **-** , ***** , / }

$E$ ⟦**E₁ rop E₂**⟧ env = let $e_1 = E$ ⟦**E₁**⟧ env and $e_2 = E$ ⟦**E₂**⟧ env
in if ((*type* $e_1 \neq$ *integer*) or (*type* $e_2 \neq$ *integer*)) then $\top$
else (*expr-val boolean* (*RO* ⟦**rop**⟧ (*value* $e_1$) (*value* $e_2$)))
end where **rop** $\in$ {**<** , **>** , **<=** , **>=**}

$E$ ⟦**E₁ == E₂**⟧ env = let $e_1 = E$ ⟦**E₁**⟧ env and $e_2 = E$ ⟦**E₂**⟧ env in
in if (*type* $e_1$ = *integer*) and (*type* $e_2$ = *integer*) then
then (*expr-val boolean* (*RO* ⟦**=**⟧ (*value* $e_1$) (*value* $e_2$)))
else if (*type* $e_1$ = *list*) and (*type* $e_2$ = *list*)
then (*expr-val boolean* (*eqlist* (*value* $e_1$) (*value* $e_2$)))
else $\top$
end

$E$ ⟦**E₁ /= E₂**⟧ env = let $e_1 = E$ ⟦**E₁**⟧ env and $e_2 = E$ ⟦**E₂**⟧ env in
in if (*type* $e_1$ = *integer*) and (*type* $e_2$ = *integer*) then
then (*expr-val boolean* (*RO* ⟦**/=**⟧ (*value* $e_1$) (*value* $e_2$)))
else if (*type* $e_1$ = *list*) and (*type* $e_2$ = *list*)
then (*expr-val boolean* (*neqlist* (*value* $e_1$) (*value* $e_2$)))
else $\top$
end

$E$ ⟦**E₁ && E₂**⟧ env =

```
    let e1 = E ⟦E1⟧ env
    in      if (type e1 ≠ boolean)  then ⊤              "false by short-circuit evaluation below"
            else    if (value e1 = false)  then (expr-val boolean (value e1))
                    else    let e2 = E ⟦E2⟧ env
                            in      if (type e2 = boolean)  then  (expr-val boolean (value e2))
                                    else ⊤
                            end
    end
```

$E$ ⟦**E₁ || E₂**⟧ env =

```
    let e1 = E ⟦E1⟧ env
    in      if (type e1 ≠ boolean)  then ⊤              "true by short-circuit evaluation below"
            else    if (value e1 = true)  then (expr-val boolean (value e1))
                    else    let e2 = E ⟦E2⟧ env
                            in      if (type e2 = boolean)  then  (expr-val boolean (value e2))
                                    else ⊤
                            end
    end
```