COBALT – Comprehension-supported Boolean and Arithmetic Expression with Lists and Tuples

1 INTRODUCTION

COBALT is a toy language for the COSE212 course at Korea University. COBALT stands for **CO**mprehension-supported **B**oolean and **A**rithmetic expressions with **L**ists and **T**uples, and it supports the following features:

- **number (integer) values** (0, 1, -1, 2, -2, 3, -3, . . .)
- **boolean values** (true and false)
- **arithmetic operators**: negation (-), addition (+), subtraction (-), multiplication (*), division (/), and modulo (%)
- **logical operators**: conjunction (&&), disjunction (||), and negation (!)
- **comparison operators**: equality (== and !=) and relational (<, >, <=, and >=)
- **conditionals** (if-else)
- **lists** (Nil, ::, and List)
- **list operations** (head, tail, isEmpty, map, flatMap, filter), and foldLeft
- **list comprehension** (for-yield)
- **tuples** ((e1, ..., en) where $n \geq 2$)
- **tuple projections** (_1, _2, . . .)
- **immutable variable definitions** (val)
- **first-class functions** (=>)
- **mutually recursive functions** (def)

This document is the specification of COBALT. First, Section 2 describes the concrete syntax, and Section 3 describes the abstract syntax with the desugaring rules. Then, Section 4 describes the big-step operational (natural) semantics of COBALT.

2 CONCRETE SYNTAX

The concrete syntax of COBALT is written in a variant of the extended Backus–Naur form (EBNF). The notation `<nt>` denotes a nonterminal, and `"t"` denotes a terminal. We use ? to denote an optional element and + (or *) to denote one or more (or zero or more) repetitions of the preceding element. We use butnot to denote a set difference to exclude some strings from a producible set of strings. We omit some obvious terminals using the ellipsis (. . .) notation.

```
// basic elements
<digit>   ::= "0" | "1" | "2" | ... | "9"
<nonzero> ::= "1" | "2" | ... | "9"
<number>  ::= "-"? <digit>+
<index>   ::= "_" <nonzero> <digit>*
<alphabet> ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart> ::= <alphabet> | "_"
<idcont>  ::= <alphabet> | "_" | <digit>
<keyword> ::= "List" | "Nil" | "def" | "else" | "false" | "for"
            | "if" | "true" | "val" | "yield"
<id>      ::= <idstart> <idcont>* butnot <keyword>
```

```
// expressions
<expr> ::= "()" | <number> | "true" | "false" | <id>
        // unary and binary operators
        | <uop> <expr> | <expr> <bop> <expr>
        // parentheses
        | "(" <expr> ")" | "{" <expr> "}"
        // conditionals
        | "if" "(" <expr> ")" <expr> "else" <expr>
        // lists and list operations
        | "Nil" | <expr> "::" <expr> | "List(" <expr> ")"
        | <expr> "." "head" | <expr> "." "tail"
        | <expr> "." "isEmpty" | <expr> "." "map" "(" <expr> ")"
        | <expr> "." "flatMap" "(" <expr> ")"
        | <expr> "." "filter" "(" <expr> ")"
        | <expr> "." "foldLeft" "(" <expr>, <expr> ")"
        // list comprehensions
        | "for" "{" <comp>+ "}" "yield" <expr>
        // tuples and tuple projections
        | "(" <expr> "," <expr> ["," <expr>]* ")"
        | <expr> "." <index>
        // first-class functions
        | "(" ")" "=>" <expr> | <id> "=>" <expr>
        | "(" <id> ["," <id>]* ")" "=>" <expr>
        // mutually recursive functions
        | <fdef>+ ";" <expr>
        // function applications
        | <expr> "(" <expr> ")"
// operators
<uop>  ::= "-" | "!"
<bop>  ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "==" | "!=" | "<" | "<=" | ">" | ">="
// comprehension elements
<comp> ::= <id> "<-" <expr> ";" ["if" <expr> ";"]*
// function definitions
<fdef> ::= "def" <id> "(" ")" "=" <expr> ";"
        | "def" <id> "(" <id> ["," <id>]* ")" "=" <expr> ";"
```

The precedence and associativity of operators are defined as follows:

Description	Operator	Precedence	Associativity
Unary	-, !	1	right
Multiplicative	*, /, %	2	left
Additive	+, -	3	
List Construction	::	4	right
Relational	<, <=, >, >=	5	left
Equality	==, !=	6	
Logical Conjunction	&&	7	
Logical Disjunction	||	8	

3 ABSTRACT SYNTAX

The abstract syntax of COBALT is defined as follows:

Expressions $\mathbb{E} \ni e ::= n$ (ENum) | Nil (ENil)

| b (EBool) | $e :: e$ (ECons)

| x (EId) | $e.\text{head}$ (EHead)

| $e + e$ (EAdd) | $e.\text{tail}$ (ETail)

| $e * e$ (EMul) | $e.\text{map}(e)$ (EMap)

| $e\ /\ e$ (EDiv) | $e.\text{flatMap}(e)$ (EFlatMap)

| $e\ \%\ e$ (EMod) | $e.\text{filter}(e)$ (EFilter)

| $e == e$ (EEq) | $e.\text{foldLeft}(e, e)$ (EFoldLeft)

| $e < e$ (ELt) | $(e, \ldots, e)$ (ETuple) (length ≥ 2)

| $\text{if } (e)\ e \text{ else } e$ (EIf) | $e.$ (EProj)

| $\text{val } x = e;\ e$ (EVal)

| $\lambda(x, \ldots, x).e$ (EFun)

| $f \ldots f\, e$ (ERec)

| $e(e, \ldots, e)$ (EApp)

where

Identifier	$x \in \mathbb{X}$	(String)	Boolean	$\mathbb{B} \ni b ::= \text{true} \mid \text{false}$	(Boolean)
Index	$i \in \mathbb{Z}^+$	(Int)	Function Definition	$\mathbb{F} \ni f ::= \text{def } x(x, \ldots, x) = e;$	(FunDef)
Number	$n \in \mathbb{Z}$	(BigInt)			

The semantics of the remaining cases are defined with the following desugaring rules:

$$\mathcal{D}[\![- e]\!] = \mathcal{D}[\![e]\!] * (\text{-1})$$
$$\mathcal{D}[\![!\ e]\!] = \text{if } (\mathcal{D}[\![e]\!]) \text{ false else true}$$
$$\mathcal{D}[\![e_1 - e_2]\!] = \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![- e_2]\!]$$
$$\mathcal{D}[\![e_1\ \&\&\ e_2]\!] = \text{if } (\mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!] \text{ else false}$$
$$\mathcal{D}[\![e_1\ ||\ e_2]\!] = \text{if } (\mathcal{D}[\![e_1]\!]) \text{ true else } \mathcal{D}[\![e_2]\!]$$
$$\mathcal{D}[\![e_1\ !=\ e_2]\!] = \mathcal{D}[\![!\ (e_1 == e_2)]\!]$$
$$\mathcal{D}[\![e_1\ <=\ e_2]\!] = \mathcal{D}[\![(e_1 < e_2)\ ||\ (e_1 == e_2)]\!]$$
$$\mathcal{D}[\![e_1\ >\ e_2]\!] = \mathcal{D}[\![!\ (e_1 <= e_2)]\!]$$
$$\mathcal{D}[\![e_1\ >=\ e_2]\!] = \mathcal{D}[\![!\ (e_1 < e_2)]\!]$$
$$\mathcal{D}[\![\text{List}(e_1, \ldots, e_n)]\!] = \mathcal{D}[\![e_1]\!] :: \ldots :: \mathcal{D}[\![e_n]\!] :: \text{Nil}$$
$$\mathcal{D}[\![e.\text{isEmpty}]\!] = \mathcal{D}[\![e]\!] == \text{Nil}$$

$$\mathcal{D}\left[\!\left[\begin{array}{l} \text{for } \{ \\ \quad x_1 \leftarrow e_1; \text{if } e_{1,1};\ \text{if } e_{1,1};\ \ldots \text{if } e_{1,k_1}; \\ \quad \ldots \\ \quad x_n \leftarrow e_n; \text{if } e_{n,1};\ \text{if } e_{n,1};\ \ldots \text{if } e_{n,k_n}; \\ \} \text{ yield } e \end{array}\right]\!\right] = \begin{array}{l} \mathcal{D}[\![e_1]\!] \\ \quad .\text{filter}(x_1 => \mathcal{D}[\![e_{1,1}]\!]) \\ \quad \ldots \\ \quad .\text{filter}(x_1 => \mathcal{D}[\![e_{1,k_1}]\!]) \\ \quad .\text{flatMap}(x_1 => \{ \\ \qquad \ldots \\ \qquad \mathcal{D}[\![e_n]\!] \\ \qquad \quad .\text{filter}(x_n => \mathcal{D}[\![e_{n,1}]\!]) \\ \qquad \quad \ldots \\ \qquad \quad .\text{filter}(x_n => \mathcal{D}[\![e_{n,k_n}]\!]) \\ \qquad \quad .\text{map}(x_n => \mathcal{D}[\![e]\!]) \\ \quad \}) \end{array}$$

The omitted cases recursively apply the desugaring rule to sub-expressions.

4 SEMANTICS

We use the following notations in the semantics:

Value $\qquad \mathbb{V} \ni v ::= n \qquad$ (NumV) $\quad | \text{ Nil} \quad$ (NilV) $\quad | (v, \ldots, v) \qquad$ (TupleV)

$\qquad\qquad\qquad\quad | \ b \qquad$ (BoolV) $\quad | \ v :: v \quad$ (ConsV) $\quad | \langle \lambda(x, \ldots, x).e, \sigma \rangle \quad$ (CloV)

Environment $\qquad \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} \quad$ (Env)

The big-step operational (natural) semantics of COBALT is defined as follows:

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\text{Num} \frac{}{\sigma \vdash n \Rightarrow n} \qquad \text{Bool} \frac{}{\sigma \vdash b \Rightarrow b} \qquad \text{Id} \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

$$\text{Add} \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \text{Mul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2}$$

$$\text{Div} \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 / e_2 \Rightarrow n_1 \div n_2} \qquad \text{Mod} \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 \% e_2 \Rightarrow n_1 \% n_2}$$

$$\text{Eq} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 == e_2 \Rightarrow \text{eq}(v_1, v_2)} \qquad \text{Lt} \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

$$\text{If}_T \frac{\sigma \vdash e_0 \Rightarrow \text{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \text{if } (e_0) \ e_1 \text{ else } e_2 \Rightarrow v_1} \qquad \text{If}_F \frac{\sigma \vdash e_0 \Rightarrow \text{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{if } (e_0) \ e_1 \text{ else } e_2 \Rightarrow v_2}$$

$$\text{Nil} \frac{}{\sigma \vdash \text{Nil} \Rightarrow \text{Nil}} \qquad \text{Cons} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad (v_2 = \text{Nil} \vee v_2 = \_ :: \_)}{\sigma \vdash e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

$$\text{Head} \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\text{head} \Rightarrow v_1} \qquad \text{Tail} \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\text{tail} \Rightarrow v_2}$$

$$\text{Map} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad \text{map}(v_1, v_2) = v'}{\sigma \vdash e_1.\text{map}(e_2) \Rightarrow v'}$$

$$\text{FlatMap} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad \text{join}(\text{map}(v_1, v_2)) = v'}{\sigma \vdash e_1.\text{flatMap}(e_2) \Rightarrow v'}$$

$$\text{Filter} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad \text{filter}(v_1, v_2) = v'}{\sigma \vdash e_1.\text{filter}(e_2) \Rightarrow v'}$$

$$\text{FoldLeft} \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad \sigma \vdash e_3 \Rightarrow v_3 \qquad \text{foldLeft}(v_1, v_2, v_3) = v'}{\sigma \vdash e_1.\text{foldLeft}(e_2, e_3) \Rightarrow v'}$$

$$\text{Tuple } \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \ldots \quad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash (e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_n)} \qquad \text{Proj } \frac{\sigma \vdash e \Rightarrow (v_1, \ldots, v_n)}{\sigma \vdash e.i \Rightarrow v_i}$$

$$\text{Val } \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; \ e_2 \Rightarrow v_2} \qquad \text{Fun } \frac{}{\sigma \vdash \lambda(x_1, \ldots, x_n).e \Rightarrow \langle \lambda(x_1, \ldots, x_n).e, \sigma \rangle}$$

$$\text{Rec } \frac{\sigma' = \sigma[x_1 \mapsto \langle \lambda(x_{1,1}, \ldots, x_{1,k_1}).e_1, \sigma' \rangle, \ldots x_n \mapsto \langle \lambda(x_{n,1}, \ldots, x_{n,k_n}).e_n, \sigma' \rangle,] \quad \sigma' \vdash e \Rightarrow v}{\sigma \vdash \text{def } x_1(x_{1,1}, \ldots, x_{1,k_1}) = e_1; \ldots \text{def } x_n(x_{n,1}, \ldots, x_{n,k_n}) = e_n; e \Rightarrow v}$$

$$\text{App } \frac{\sigma \vdash e_0 \Rightarrow v_0 \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \ldots \quad \sigma \vdash e_n \Rightarrow v_n \quad \text{app}(v_0, [v_1, \ldots, v_n]) = v}{\sigma \vdash e_0(e_1, \ldots, e_n) \Rightarrow v}$$

with the following auxiliary partial functions, and $X \rightharpoonup Y$ denotes a partial function from X to Y.

$$\boxed{\text{eq} : \mathbb{V} \times \mathbb{V} \to \mathbb{B}}$$

$$\begin{array}{llll}
\text{eq}(n, n) & = \text{true} & \text{eq}(\text{Nil}, \_ :: \_) & = \text{false} \\
\text{eq}(b, b) & = \text{true} & \text{eq}(\_ :: \_, \text{Nil}) & = \text{false} \\
\text{eq}(\text{Nil}, \text{Nil}) = \text{true} & & \text{eq}(v_h :: v_t, v_h' :: v_t') = \text{eq}(v_h, v_h') \wedge \text{eq}(v_t, v_t')
\end{array}$$

Note that eq is a total function and should return false for the cases not listed above.

$$\boxed{\text{map} : \mathbb{V} \times \mathbb{V} \rightharpoonup \mathbb{V}} \qquad\qquad\qquad \boxed{\text{join} : \mathbb{V} \rightharpoonup \mathbb{V}}$$

$$\begin{array}{ll}
\text{map}(\text{Nil}, \_) = \text{Nil} & \qquad \text{join}(\text{Nil}) = \text{Nil} \\
\text{map}(v_h :: v_t, v_f) = \text{app}(v_f, [v_h]) :: \text{map}(v_t, v_f) & \qquad \text{join}(\text{Nil} :: v_t) = \text{join}(v_t) \\
& \qquad \text{join}((v_h :: v_t) :: v_t') = v_h :: \text{join}(v_t :: v_t')
\end{array}$$

$$\boxed{\text{filter} : \mathbb{V} \times \mathbb{V} \rightharpoonup \mathbb{V}}$$

$$\text{filter}(\text{Nil}, \_) = \text{Nil}$$
$$\text{filter}(v_h :: v_t, v_f) = \begin{cases} v_h :: \text{filter}(v_t, v_f) & \text{if } \text{app}(v_f, [v_h]) = \text{true} \\ \text{filter}(v_t, v_f) & \text{if } \text{app}(v_f, [v_h]) = \text{false} \end{cases}$$

$$\boxed{\text{foldLeft} : \mathbb{V} \times \mathbb{V} \times \mathbb{V} \rightharpoonup \mathbb{V}}$$

$$\text{foldLeft}(\text{Nil}, v_i, \_) = v_i$$
$$\text{filter}(v_h :: v_t, v_i, v_f) = \text{foldLeft}(v_t, \text{app}(v_f, [v_i, v_h]), v_f)$$

$$\boxed{\text{app} : \mathbb{V} \times \mathbb{V}^* \rightharpoonup \mathbb{V}}$$

$$\text{app}(\langle \lambda(x_1, \ldots, x_n).e, \sigma \rangle, [v_1, \ldots, v_n]) = v \quad \text{where} \quad \sigma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e \Rightarrow v$$