

# 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 **C**omprehension-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)	$(e, \dots, e)$	(ETuple) (length $\geq 2$ )
	if $(e) e$ else $e$	(EIf)	$e.$	(EProj)
			val $x = e; e$	(EVal)
			$\lambda(x, \dots, x).e$	(EFun)
			$f \dots fe$	(ERec)
			$e(e, \dots, e)$	(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	$\mathbb{F} \ni f ::= \text{def } x(x, \dots, x) = e;$	(FunDef)
Number	$n \in \mathbb{Z}$	(BigInt)			

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

$$\begin{aligned}
\mathcal{D}[-e] &= \mathcal{D}[e] * (-1) \\
\mathcal{D}[\neg e] &= \text{if } (\mathcal{D}[e]) \text{ false else true} \\
\mathcal{D}[e_1 - e_2] &= \mathcal{D}[e_1] + \mathcal{D}[-e_2] \\
\mathcal{D}[e_1 \&\& e_2] &= \text{if } (\mathcal{D}[e_1]) \mathcal{D}[e_2] \text{ else false} \\
\mathcal{D}[e_1 \parallel e_2] &= \text{if } (\mathcal{D}[e_1]) \text{ true else } \mathcal{D}[e_2] \\
\mathcal{D}[e_1 \neq e_2] &= \mathcal{D}[\neg (e_1 == e_2)] \\
\mathcal{D}[e_1 < e_2] &= \mathcal{D}[(e_1 < e_2) \parallel (e_1 == e_2)] \\
\mathcal{D}[e_1 > e_2] &= \mathcal{D}[\neg (e_1 < e_2)] \\
\mathcal{D}[e_1 \geq e_2] &= \mathcal{D}[\neg (e_1 < e_2)] \\
\mathcal{D}[\text{List}(e_1, \dots, e_n)] &= \mathcal{D}[e_1] :: \dots :: \mathcal{D}[e_n] :: \text{Nil} \\
\mathcal{D}[e.\text{isEmpty}] &= \mathcal{D}[e] == \text{Nil}
\end{aligned}$$

$$\mathcal{D} \left[ \begin{array}{l} \text{for } \{ \\ \quad x_1 <- e_1; \text{if } e_{1,1}; \text{if } e_{1,2}; \dots \text{if } e_{1,k_1}; \\ \quad \dots \\ \quad x_n <- e_n; \text{if } e_{n,1}; \text{if } e_{n,2}; \dots \text{if } e_{n,k_n}; \\ \} \text{yield } e \end{array} \right] = \begin{array}{l} \mathcal{D}[e_1] \\ \quad .\text{filter}(x_1 \Rightarrow \mathcal{D}[e_{1,1}]) \\ \quad \dots \\ \quad .\text{filter}(x_1 \Rightarrow \mathcal{D}[e_{1,k_1}]) \\ \quad .\text{flatMap}(x_1 \Rightarrow \{ \\ \quad \quad \dots \\ \quad \quad \mathcal{D}[e_n] \\ \quad \quad \quad .\text{filter}(x_n \Rightarrow \mathcal{D}[e_{n,1}]) \\ \quad \quad \quad \dots \\ \quad \quad \quad .\text{filter}(x_n \Rightarrow \mathcal{D}[e_{n,k_n}]) \\ \quad \quad \quad .\text{map}(x_n \Rightarrow \mathcal{D}[e]) \\ \quad \quad \}) \end{array}$$

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



$$\begin{array}{c}
\text{Tuple} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash (e_1, \dots, e_n) \Rightarrow (v_1, \dots, v_n)} \quad \text{Proj} \frac{\sigma \vdash e \Rightarrow (v_1, \dots, v_n)}{\sigma \vdash e.i \Rightarrow v_i} \\
\\
\text{Val} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; e_2 \Rightarrow v_2} \quad \text{Fun} \frac{}{\sigma \vdash \lambda(x_1, \dots, x_n).e \Rightarrow \langle \lambda(x_1, \dots, x_n).e, \sigma \rangle} \\
\\
\text{Rec} \frac{\sigma' = \sigma[x_1 \mapsto \langle \lambda(x_{1,1}, \dots, x_{1,k_1}).e_1, \sigma' \rangle, \dots, x_n \mapsto \langle \lambda(x_{n,1}, \dots, x_{n,k_n}).e_n, \sigma' \rangle, ] \quad \sigma' \vdash e \Rightarrow v}{\sigma \vdash \text{def } x_1(x_{1,1}, \dots, x_{1,k_1}) = e_1; \dots \text{def } x_n(x_{n,1}, \dots, x_{n,k_n}) = e_n; e \Rightarrow v} \\
\\
\text{App} \frac{\sigma \vdash e_0 \Rightarrow v_0 \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n \quad \text{app}(v_0, [v_1, \dots, v_n]) = v}{\sigma \vdash e_0(e_1, \dots, e_n) \Rightarrow v}
\end{array}$$

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

$$\text{eq} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{B}$$

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

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

$$\text{map} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$$

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

$$\text{join} : \mathbb{V} \rightarrow \mathbb{V}$$

$$\begin{array}{ll}
\text{join}(\text{Nil}) &= \text{Nil} \\
\text{join}(\text{Nil} :: v_t) &= \text{join}(v_t) \\
\text{join}((v_h :: v_t) :: v'_t) &= v_h :: \text{join}(v_t :: v'_t)
\end{array}$$

$$\text{filter} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$$

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

$$\text{foldLeft} : \mathbb{V} \times \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$$

$$\begin{array}{ll}
\text{foldLeft}(\text{Nil}, v_i, \_) &= v_i \\
\text{foldLeft}(v_h :: v_t, v_i, v_f) &= \text{foldLeft}(v_t, \text{app}(v_f, [v_i, v_h]), v_f)
\end{array}$$

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

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