Oliver Graf

# The

# Dromedar

## Programming Language

# Contents


**1 Introduction** **3**

**2 Typing System** **3**

**3 Expressions** **3**

**4 Handling Whitespace** **4**

**5 Buildup of a Program** **4**

**6 Formal Typing Rules** **4**

**7 Formal Grammar** **9**

# 1 Introduction

# 2 Typing System

Dromedar uses a static, strong and sound typing system – typing errors cannot happen at runtime. It knows primitive and reference types.

## 2.1 Primitives

There are four primitive types:

- **`int`** represents 64-bit 2's complement (signed) integers.
- **`flt`** represents 64-bit IEEE-754 standard floating point numbers.
- **`char`** represents 8-bit UTF-8 characters.
- **`bool`** represents a 1-bit value: **`true`** and **`false`**.

## 2.2 Reference Types

Reference Types of type `t` come in two shapes: `t`, a strictly non-`null` reference; and `t?`, a reference that may be `null`, where `t` is considered a subtype of `t?`.

Splitting up references that way has two advantages: On one hand, it enforces a good coding style and helps find Null pointer bugs before they even appear; on the other hand it accelerates the execution environment by allowing it to skip Null checks when dereferencing an object if it is of a strictly non-`null` type.

# 3 Expressions

Because Dromedar has a strong type system, it generally disallows any operations with operands of a non-specified type unless they are explicitly cast to the correct type before. This means that integer and floating point numbers cannot be added, multiplied, etc.

The following table describes precedence and type of all operators:

| Operator | Name | Prec. | Assoc. | Types |
|---|---|---|---|---|
| `-`<br><br>`!` | Unary Negation<br><br>Logical Negation | 100 | non-assoc. | **int** -> **int**<br>**flt** -> **flt**<br>**bool** -> **bool** |
| `**` | Exponentiation | 90 | right | **int,int** -> **int**<br>**flt,flt** -> **flt** |
| `*` | Multiplication | 80 | left | **int,int** -> **int**<br>**flt,flt** -> **flt** |
| `+,-` | Addition<br>Subtraction | 70 | left | **int,int** -> **int**<br>**flt,flt** -> **flt**<br>**char,int** -> **char**<br>**int,char** -> **char** |
| `<<,>>,>>>` | Left Shift<br>Logical Right Shift<br>Arithmetic Right Shift | 60 | left | **int,int** -> **int** |
| `&` | Bitwise And | 60 | left | **int,int** -> **int** |
| `^` | Bitwise Xor | 50 | left | **int,int** -> **int** |
| `\|` | Bitwise Or | 40 | left | **int,int** -> **int** |
| `=,!=,>,<,>=,<=` | Comparison | 30 | non-assoc. | [**int**] -> **bool**<br>[**flt**] -> **bool** |
| `&&` | Logical And | 20 | left | **bool,bool** -> **bool** |
| `\|\|` | Logical Or | 10 | left | **bool,bool** -> **bool** |

Consider the following example:

The expression `1+18-18+'a'` is well-typed (of type **`char`**) and gets parsed as `((1+18)-18)+'a'` and evaluated to `'b'`. `10 - 0.0` on the other hand is not well-typed as `-` cannot take an **`int`** and a **`flt`** operand as arguments.

Comparison operators work differently to other (binary) operators: Instead of comparing just two expressions, Dromedar allows chaining expressions to create one final boolean value: For example, `1 < 2 != 5 >= 5` holds because every single sub-expression (`1 < 2`, `2 != 5` and `5 >= 5`) holds. Every expression is only evaluated once for its side-effect.

Thus, `A op B op C` is not necessarily semantically equivalent to `A op B && B op C`.

# 4 Handling Whitespace

In order to keep the code simple and easy to look at, Dromedar uses significant whitespace: Blocks of code (such as bodies of **`if`**) statements, are denoted by adding a level of indentation.

Every line is either empty (this includes lines containing only comments), or it contains code.

If a line contains code, the level of its indentation is determined by its relationship to the previous line and its environment:

Two neighboring lines of code within the same block of code must have exactly matching whitespace characters before their respective code starts. If a following line has a deeper level of indentation, it must match the whitespace characters of the previous line and then add a number of additional whitespace characters (space(s) and/or tab(s)).

A line of code can only have a deeper indentation level of one step compared to the previous line. The first line of code is always a global instruction and as such has the lowest level of indentation. If it is indented, this level of indentation corresponds to a baseline indentation that every line of code must share.

Consider this valid example:

```
  global x := 3         # baseline indentation of two spaces
                        # empty line -> indentation doesn't matter
  fn main : args:[string] -> int
          Stdio.println("Hello, World!")  # deeper indentation level
          return 0                        # same indentation level
```

Blocks with the same level of indentation can have different indentation strings, but they must still match their environments, as follows:

```
       if <condition>
             BLOCK A
       else
         BLOCK B
```

The two blocks have a different indentation level, but from context it is still clear that `BLOCK A` is a sub-block of the **`if`**-statement, whereas `BLOCK B` belongs to the **`else`**-statement.

# 5 Buildup of a Program

A program consists of a series of global statements – global variable declarations and function definitions.

## 5.1 Global Declarations

Global variables are all assigned a value at the point of their declaration. This value is evaluated statically – declaration expressions can only contain literals and global variables that were already declared previously.

Functions are also declared globally.

# 6 Formal Typing Rules

A typing rule takes the following shape:

$$\frac{\text{Hypotheses}}{S, \cdots \vdash \texttt{grammar spec}}\textsc{name}$$

Here, $S$ represents a stack of symbol definitions: $S \in (\mathsf{id} \times \mathsf{type})^n$ for some block depth $n$ at any given point. In global context, $S$ has only one layer. id corresponds to the set of names that variables can have (related to the Lexer symbol `%Identifier`), and type to the set of types in a given program (related to the Parser symbol `Type`).

The symbol $\in$ is defined as follows: $s \in S \Leftrightarrow s$ is contained in *any* layer of $S$, whereas $\in_0$ is true only if the symbol is at the top level of the symbol stack (i.e. defined in the same block). The operator $\cup$ on $S$ adds another binding to the top layer of the stack, whereas $\sqcup$ adds another layer to the stack.

The following are the typing rules for Dromedar programs:

## 6.1 Builtin Operators

Many builtin operators are overloaded, providing functionality for multiple input types.

### 6.1.1 Unary Operators

#### 6.1.1.1 Arithmetic Negation

$$\frac{}{\vdash_O \texttt{-} :: \mathsf{int} \to \mathsf{int}}\textsc{TyUopNeg}, \quad \frac{}{\vdash_O \texttt{-} :: \mathsf{flt} \to \mathsf{flt}}\textsc{TyUopNeg}$$

#### 6.1.1.2 Logical Negation

$$\frac{}{\vdash_O \texttt{!} :: \mathsf{bool} \to \mathsf{bool}}\textsc{TyUopNot}$$

### 6.1.2 Binary Operators

#### 6.1.2.1 Power

$$\frac{}{\vdash_O \texttt{**} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopPow}, \quad \frac{}{\vdash_O \texttt{**} :: (\mathsf{flt}, \mathsf{flt}) \to \mathsf{flt}}\textsc{tyBopPow}$$

#### 6.1.2.2 Multiplication

$$\frac{}{\vdash_O \texttt{*} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopMul}, \quad \frac{}{\vdash_O \texttt{*} :: (\mathsf{flt}, \mathsf{flt}) \to \mathsf{flt}}\textsc{TyBopMul}$$

#### 6.1.2.3 Addition and Subtraction

- $$\frac{}{\vdash_O \texttt{+} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopAdd}, \quad \frac{}{\vdash_O \texttt{+} :: (\mathsf{flt}, \mathsf{flt}) \to \mathsf{flt}}\textsc{TyBopAdd},$$
$$\frac{}{\vdash_O \texttt{+} :: (\mathsf{int}, \mathsf{char}) \to \mathsf{char}}\textsc{TyBopAdd}, \quad \frac{}{\vdash_O \texttt{+} :: (\mathsf{char}, \mathsf{int}) \to \mathsf{char}}\textsc{TyBopAdd}$$

- $$\frac{}{\vdash_O \texttt{-} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopSub}, \quad \frac{}{\vdash_O \texttt{-} :: (\mathsf{flt}, \mathsf{flt}) \to \mathsf{flt}}\textsc{TyBopSub},$$
$$\frac{}{\vdash_O \texttt{-} :: (\mathsf{int}, \mathsf{char}) \to \mathsf{char}}\textsc{TyBopSub}, \quad \frac{}{\vdash_O \texttt{-} :: (\mathsf{char}, \mathsf{int}) \to \mathsf{char}}\textsc{TyBopSub}$$

#### 6.1.2.4 Shift Operators

- $$\frac{}{\vdash_O \texttt{<<} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopLShift}$$

- $$\frac{}{\vdash_O \texttt{>>} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopBitRShift}$$

- $$\frac{}{\vdash_O \texttt{>>>} :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopBitAShift}$$

#### 6.1.2.5 Bitwise Operators

- $$\frac{}{\vdash_O \ \&\ :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopBitAnd}$$

- $$\frac{}{\vdash_O \ \hat{}\ :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopXor}, \quad \frac{}{\vdash_O \ \hat{}\ :: (\mathsf{bool}, \mathsf{bool}) \to \mathsf{bool}}\textsc{TyBopXor}$$

- $$\frac{}{\vdash_O \ |\ :: (\mathsf{int}, \mathsf{int}) \to \mathsf{int}}\textsc{TyBopBitOr}$$

#### 6.1.2.6 Logical Operators

- $$\frac{}{\vdash_O \ \&\&\ :: (\mathsf{bool}, \mathsf{bool}) \to \mathsf{bool}}\textsc{TyBopLogAnd}$$

- $$\frac{}{\vdash_O \ ||\ :: (\mathsf{bool}, \mathsf{bool}) \to \mathsf{bool}}\textsc{TyBopLogOr}$$

#### 6.1.2.7 Comparison Operators

$$\frac{}{\vdash_O \{=, !=, >, <, >=, <=\} :: [\mathsf{int}] \to \mathsf{bool}}\textsc{TyCmpList}, \quad \frac{}{\vdash_O \{=, !=, >, <, >=, <=\} :: [\mathsf{flt}] \to \mathsf{bool}}\textsc{TyCmpList},$$
$$\frac{}{\vdash_O \{=, !=, >, <, >=, <=\} :: [\mathsf{char}] \to \mathsf{bool}}\textsc{TyCmpList},$$

## 6.2 Expressions

### 6.2.1 Assignability

The rule $\vdash_A$ defines when an expression can be assigned a value.

$$\frac{(\mathtt{id}, \mathsf{t}) \in S}{S \vdash_A \mathtt{id}}\textsc{ExpAssn}$$

### 6.2.2 Global Expressions

The rule $\vdash_G$ describes global expressions, which are restricted in a way that they can be computed at compile time.

Copied rules always use $\vdash_G$ instead of $\vdash_E$

#### 6.2.2.1 Literals

- GExpLitInt := ExpLitInt
- GExpLitFlt := ExpLitFlt
- GExpLitChar := ExpLitChar
- GExpLitBool := ExpLitBool

#### 6.2.2.2 Other Rules

- GExpId := ExpId
- GExpUop := ExpUop
- GExpBop := ExpBop
- GExpCmpList := ExpCmpList

Note that global variable declarations cannot feature function calls.

### 6.2.3 Literals

- $$\frac{}{\vdash_{\mathsf{E}} n :: \textbf{int}}\textsc{ExpLitInt}$$

- $$\frac{}{\vdash_{\mathsf{E}} f :: \textbf{flt}}\textsc{ExpLitFlt}$$

- $$\frac{}{\vdash_{\mathsf{E}} c :: \textbf{char}}\textsc{ExpLitChar}$$

- $$\frac{}{\vdash_{\mathsf{E}} \textbf{true} :: \textbf{bool}}\textsc{ExpLitBool}, \quad \frac{}{\vdash_{\mathsf{E}} \textbf{true} :: \textbf{bool}}\textsc{ExpLitBool}$$

### 6.2.4 Identifiers

$$\frac{(id, \mathsf{t}) \in S}{S \vdash_{\mathsf{E}} \mathtt{id} :: \mathsf{t}}\textsc{ExpId}$$

### 6.2.5 Unary Operations

$$\frac{S \vdash \mathsf{exp} :: \mathsf{t}' \quad S \vdash \mathsf{op} :: \mathsf{t}' \to \mathsf{t}}{S \vdash_{\mathsf{E}} \mathtt{op\ \ exp} :: \mathsf{t}}\textsc{ExpUop}$$

### 6.2.6 Binary Operations

$$\frac{S \vdash \mathtt{e1} :: \mathsf{t1} \quad S \vdash \mathtt{e2} :: \mathsf{t2} \quad S \vdash \mathsf{op} :: (\mathsf{t1}, \mathsf{t2}) \to \mathsf{t}}{S \vdash_{\mathsf{E}} \mathtt{e1\ \ op\ \ e2} :: \mathsf{t}}\textsc{ExpBop}$$

### 6.2.7 Function Calls

$$\frac{S \vdash \mathsf{f} :: (\mathsf{t1}, ..., \mathsf{tn}) \to \mathsf{rt} \quad S \vdash \mathtt{e1} :: \mathsf{t1}, \ \ldots, \ S \vdash \mathsf{en} :: \mathsf{tn}}{S \vdash_{\mathsf{E}} \mathtt{f(e1,...,en)} :: \mathsf{rt}}\textsc{ExpFunc}$$

### 6.2.8 Comparison Lists

$$\frac{S \vdash \mathtt{e0} :: \mathsf{t0}, \ \ldots, \ S \vdash \mathsf{en} :: \mathsf{tn} \quad S \vdash \mathtt{op1} :: (\mathsf{t0}, \mathsf{t1}) \to \mathsf{t}, \ \ldots, \ S \vdash \mathsf{opn} :: (\mathsf{t(n-1)}, \mathsf{tn}) \to \mathsf{t}}{S \vdash_{\mathsf{E}} \mathtt{e0\ \ op1\ \ ...\ \ opn\ \ en} :: \mathsf{t}}\textsc{ExpCmpList}$$

## 6.3 Statements

In the statement typing rules, a statement rule produces a tuple $(S, r)$ where $S$ stands for the newly updates context, and $r \in \{\perp, \top\}$, where $\perp$ means that a statement might not return and $\top$ means that a statement might return.

To prevent potential mistakes, the typechecker prevents statements which are deemed unreachable at compile time. Logical operators like $\wedge$ operate as if $\perp \equiv 0$ and $\top \equiv 1$.

### 6.3.1 Local Variable Declarations

$$\frac{S \vdash \mathsf{exp} :: \mathsf{t}}{S, \mathsf{rt} \vdash_{\mathsf{S}} \mathtt{let\ \ id\ \ :=\ \ exp} \Rightarrow (S \cup (\mathtt{id}, \mathsf{t})), \perp}\textsc{StmtVDecl}$$

### 6.3.2 Assignments

$$\frac{S \vdash \mathsf{exp} :: \mathsf{t} \quad S \vdash \mathtt{lhs} :: \mathsf{t} \quad S \vdash_A \mathtt{lhs}}{S, \mathsf{rt} \vdash_{\mathsf{S}} \mathtt{lhs\ \ :=\ \ exp} \Rightarrow S, \bot}\textsc{StmtAssn}$$

### 6.3.3 Expression Statements

$$\frac{S \vdash \mathsf{exp} :: \mathsf{t}}{S, \mathsf{rt} \vdash_{\mathsf{S}} \mathsf{exp} \Rightarrow S, \bot}\textsc{StmtExpr}$$

### 6.3.4 If Statements

$$\frac{S \vdash \mathsf{c} :: \textbf{bool} \quad S, \mathsf{rt} \vdash \mathsf{b1} \Rightarrow S_1, R_1 \quad S, \mathsf{rt} \vdash \mathsf{b2} \Rightarrow S_2, R_2}{S, \mathsf{rt} \vdash_{\mathsf{S}} \textbf{if}\ \ \mathtt{c}\ \ \mathtt{b1}\ \ \textbf{else}\ \ \mathtt{b2} \Rightarrow S, R_1 \wedge R_2}\textsc{StmtIf}$$

### 6.3.5 While Statements

$$\frac{S \vdash \mathsf{c} :: \textbf{bool} \quad S, \mathsf{rt} \vdash \mathsf{b} \Rightarrow S', R}{S, \mathsf{rt} \vdash_{\mathsf{S}} \textbf{while}\ \ \mathtt{c}\ \ \mathtt{b} \Rightarrow S, \bot}\textsc{StmtWhile}$$

### 6.3.6 Do-While Statements

$$\frac{S \vdash \mathsf{c} :: \textbf{bool} \quad S, \mathsf{rt} \vdash \mathsf{b} \Rightarrow S', R}{S, \mathsf{rt} \vdash_{\mathsf{S}} \textbf{do}\ \ \mathtt{b}\ \ \textbf{while}\ \ \mathtt{c} \Rightarrow S, R}\textsc{StmtDoWhile}$$

### 6.3.7 Return Statements

$$\frac{S \vdash \mathsf{exp} :: \mathsf{rt}}{S, \mathsf{rt} \vdash_{\textbf{return}\ \mathsf{exp} \Rightarrow \mathsf{S}, \top} \mathit{StmtReturn}}, \quad \frac{}{S, \textbf{void} \vdash_{\mathsf{S}} \textbf{return} \Rightarrow S, \top}\textsc{StmtReturn}$$

### 6.3.8 Blocks

$$\frac{S \sqcup \{\}, \mathsf{rt} \vdash \mathtt{s1} \Rightarrow S_1, \bot, \quad S_1, \mathsf{rt} \vdash \mathtt{s2} \Rightarrow S_2, \bot, \ \ldots, \ S_{n-1}, \mathsf{rt} \vdash \mathtt{sn} \Rightarrow S_n, R}{S, \mathsf{rt} \vdash_{\mathsf{S}} \mathtt{s1\ \ ...\ \ sn} \Rightarrow S_n, R}\textsc{StmtBlock}$$

## 6.4 Global Statements

### 6.4.1 Global Function Declaration

$$\frac{S \sqcup \{(\mathtt{a1}, \mathtt{t1}), \ldots, (\mathtt{an}, \mathtt{tn})\}, \mathtt{rt} \vdash \mathsf{b}, \top}{S \vdash_{\mathsf{G}} \textbf{fn}\ \mathtt{id\ :\ a1{:}t1,\ ...,\ an{:}tn\ \text{->}\ rt\ b} \Rightarrow S}\textsc{GStmtFDecl}$$

### 6.4.2 Global Variable Declaration

$$\frac{(\mathtt{id}, \mathit{any}) \notin S \quad S \vdash_G \mathsf{exp} :: \mathsf{t}}{S \vdash_{\mathsf{G}} \mathtt{global\ \ id\ \ :=\ \ exp} \Rightarrow S \cup \{(\mathtt{id}, \mathsf{t})\}}\textsc{GStmtVDecl}$$

### 6.4.3 Program

$$\frac{S_0 \vdash \mathtt{gs1} \Rightarrow S_1, \ \ldots, \ S_{n-1} \vdash \mathtt{gsn} \Rightarrow S_n}{S_0 := \{\} \sqcup \{\} \vdash_{\mathsf{G}} \mathtt{gs1\ \ ...\ \ gsn} \Rightarrow S_n}\textsc{GStmtProgram}$$

### 6.5 Context Type-Checking

#### 6.5.1 Global Function Declaration

$$\frac{(\texttt{id}, \mathit{any}) \notin S}{S \vdash_{\mathsf{GCF}} \texttt{\textbf{fn} id : a1:t1, ..., an:tn -> rt b} \Rightarrow S \cup (\texttt{id}, (\texttt{t1}, \ldots, \texttt{tn}) \rightarrow \texttt{rt})}\textsc{GStmtCtxtFDecl}$$

$$\frac{}{S \vdash_{\mathsf{GCF}} \texttt{let id := exp} \Rightarrow S}\textsc{GStmtCtxtFDecl}$$

#### 6.5.2 Program: Functions

$$\frac{S_0 \vdash_{\mathsf{GCF}} \texttt{gs1} \Rightarrow S_1,\ \ldots,\ S_{n-1} \vdash_{\mathsf{GCF}} \texttt{gs(n-1)} \Rightarrow S_n}{S_0 \vdash_{\mathsf{GC}} \texttt{gs1 ... gsn} \Rightarrow S_n}\textsc{GStmtCtxtFuncs}$$

#### 6.5.3 Program

$$\frac{\{\} \sqcup \{\} \vdash_{\mathsf{GCF}} \mathsf{prog} \Rightarrow S' \quad S' \vdash_{\mathsf{G}} \mathsf{prog} \Rightarrow S}{\vdash_{\mathsf{G}} \mathsf{prog} \Rightarrow S}\textsc{Prog}$$

## 7 Formal Grammar

### 7.1 Lexer Grammar

The Lexer Grammar is specified using **regular expressions**:

```
LiteralInt  ::= /[1-9]\d*/
LiteralFlt  ::= /\d+\.\d+/
LiteralChar ::= /'([^'\\]|(\\[\\nrt']))'/
LiteralBool ::= /true|false/

Identifier  ::= /[a-zA-Z][a-zA-Z0-9_]*/

global      ::= /global/
fn          ::= /fn/
let         ::= /let/
mut         ::= /mut/

int         ::= /int/
flt         ::= /flt/
char        ::= /char/
bool        ::= /bool/
void        ::= /void/

if          ::= /if/
elif        ::= /elif/
else        ::= /else/
do          ::= /do/
while       ::= /while/

return      ::= /return/

Dash        ::= /\-/
Bang        ::= /!/
Star        ::= /\*/
Plus        ::= /\+/
LShift      ::= /<</
RShift      ::= />>/
AShift      ::= />>>/
```

```
Bitand      ::= /&/
Xor         ::= /\^/
Bitor       ::= /\|/
Logand      ::= /&&/
Logor       ::= /\|\|/

Equal       ::= /=/
NotEqual    ::= /!=/
Greater     ::= />/
Less        ::= /</
GreaterEq   ::= />=/
LessEq      ::= /<=/

Assign      ::= /:=/

Colon       ::= /:/
Arrow       ::= /\->/

Comma       ::= /\,/

LParen      ::= /\(/
RParen      ::= /\)/
```

### 7.2 Parser Grammar

The following grammar specification uses a preceding `%` for lexer tokens.

```
Program          ::=
                 | €
                 | GlobalStatement Program

GlobalStatement  ::=
                 | GVDeclaration
                 | GFDeclaration

GVDeclaration    ::=
                 | %Global %Identifier %Assign GlobalExpression

GFDeclaration    ::=
                 | %Fn %Identifier                   %Arrow ReturnType Block
                 | %Fn %Identifier %Colon FArguments %Arrow ReturnType Block

FArguments       ::=
                 | %Identifier %Colon Type
                 | %Identifier %Colon Type %Comma FArguments

ReturnType       ::=
                 | %Void
                 | Type

Block            ::=
                 | €
                 | Statement Block

Statement        ::=
                 | VDeclaration
                 | AssignStmt
                 | IfStmt
                 | WhileStmt
                 | DoWhileStmt
                 | ExprStmt
```

```
                  | ReturnStmt

VDeclaration      ::=
                  | %Let %Identifier %Assign Expression

AssignStmt        ::=
                  | LHS %Assign Expression

IfStmt            ::=
                  | %If Expression Block ElifStmt

ElifStmt          ::=
                  | %Elif Expression Block ElifStmt
                  | %Else Block
                  | €

WhileStmt         ::=
                  | %While Expression Block

DoWhileStmt       ::=
                  | %Do Block %While Expression

ExprStmt          ::=
                  | Expression
                  | Expression %Colon FArgs

ReturnStmt        ::=
                  | %Return
                  | %Return Expression

LHS               ::=
                  | %Identifier

Type              ::=
                  | %Int
                  | %Flt
                  | %Char
                  | %Bool

GlobalExpression ::= Expression*

Expression        ::=
                  | ExprPrec30

ExprPrec20        ::=
                  | ExprPrec30
                  | ExprPrec30 %Equal     ExprPrec20
                  | ExprPrec30 %NotEqual  ExprPrec20
                  | ExprPrec30 %Greater   ExprPrec20
                  | ExprPrec30 %Less      ExprPrec20
                  | ExprPrec30 %GreaterEq ExprPrec20
                  | ExprPrec30 %LessEq    ExprPrec20
ExprPrec30        ::=
                  | ExprPrec40
                  | ExprPrec40 %Bitor ExprPrec30
ExprPrec40        ::=
                  | ExprPrec50
                  | ExprPrec50 %Xor ExprPrec40
ExprPrec50        ::=
                  | ExprPrec60
                  | ExprPrec60 %Bitand ExprPrec50
```

```
ExprPrec60       ::=
                 | ExprPrec70
                 | ExprPrec70 %LShift ExprPrec60
                 | ExprPrec70 %RShift ExprPrec60
                 | ExprPrec70 %AShift ExprPrec60
ExprPrec70       ::=
                 | ExprPrec80
                 | ExprPrec80 %Plus ExprPrec70
                 | ExprPrec80 %Minus ExprPrec70
ExprPrec80       ::=
                 | ExprPrec90
                 | ExprPrec90 %Star ExprPrec80
ExprPrec90       ::=
                 | ExprPrec100
                 | ExprPrec100 %StarStar ExprPrec90
ExprPrec100      ::=
                 | SimpleExpression
                 | %Dash ExprPrec100
                 | %Bang ExprPrec100
SimpleExpression ::=
                 | %LParen Expression %RParen
                 | %LiteralInt
                 | %LiteralFlt
                 | %LiteralChar
                 | %LiteralBool
                 | %Identifier
                 | %Identifier Application

Application      ::=
                 | %LParen FArgs %RParen

FArgs            ::=
                 | €
                 | NonEmptyFArgs
NonEmptyFArgs    ::=
                 | Expression
                 | Expression %Comma NonEmptyFArgs
```