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 **5**

6 Formal Typing Rules **5**

7 Formal Grammar **12**

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 represent data structures that are laid over pointers to objects stored in the heap. References come in two types: Maybe-**`null`** and Definitely-not-**`null`** types. Operations like array subscript access are only possible with non-**`null`** types to ensure **`null`** safety.

With a non-null-type `t`, the type `t?` represents a reference type that allows **`null`** values. Primitives are non-nullable, as are **`string`**s.

The following are reference types:

- **`string`** represents lists of characters.
- `[t]` represents a list of type `t`.

Thus, e.g. `[[`**`int`**`]?]` represents a two-dimensional array of integers which is definitely non-**`null`**, whereas its rows may be **`null`**.

2.3 Mutability

In general, variables declared with the **`let`** keyword are immutable, whereas **`mut`** declarations allow mutable variables.

For references, there are different notions of mutability: Allowing the object to point to new objects is handled with the **`mut`** declaration. However, even immutable objects are allowed to call methods which alter their internal state – for example changing an array's element.

2.4 Subtyping

Generally, variables and objects can be assigned values of *subtypes*. Every type is a subtype of itself, and e.g. `t` is a subtype of `t?`. Generally, a subtype is a restricted value set of its supertype – any subtype expression can be assigned to a variable of its supertype.

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:

<table>
<tr><th>Operator</th><th>Name</th><th>Prec.</th><th>Assoc.</th><th>Types</th></tr>
<tr><td>-

!</td><td>Unary Negation

Logical Negation</td><td>100</td><td>non-assoc.</td><td>int -> int
flt -> flt
bool -> bool</td></tr>
<tr><td>**</td><td>Exponentiation</td><td>90</td><td>right</td><td>int,int -> int
flt,flt -> flt</td></tr>
<tr><td>*</td><td>Multiplication</td><td>80</td><td>left</td><td>int,int -> int
flt,flt -> flt</td></tr>
<tr><td>+,-</td><td>Addition
Subtraction</td><td>70</td><td>left</td><td>int,int -> int
flt,flt -> flt
char,int -> char
int,char -> char</td></tr>
<tr><td><<,>>,>>></td><td>Left Shift
Logical Right Shift
Arithmetic Right Shift</td><td>60</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>&</td><td>Bitwise And</td><td>60</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>^</td><td>Bitwise Xor</td><td>50</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>|</td><td>Bitwise Or</td><td>40</td><td>left</td><td>int,int -> int</td></tr>
<tr><td>=,!=,>,<,>=,<=</td><td>Comparison</td><td>30</td><td>non-assoc.</td><td>[int] -> bool
[flt] -> bool</td></tr>
<tr><td>&&</td><td>Logical And</td><td>20</td><td>left</td><td>bool,bool -> bool</td></tr>
<tr><td>||</td><td>Logical Or</td><td>10</td><td>left</td><td>bool,bool -> bool</td></tr>
</table>

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. They can call each other and themselves recursively within their respective function bodies.

5.2 Builtin Functions

The Dromedar standard library includes the following functions:

5.2.1 String Operations

Name	Type	Effect
`string_of_int`	**int** -> **string**	transforms an integer into a string
`string_of_flt`	**flt** -> **string**	transforms a decimal number into a string
`string_concat`	(**string**,**string**) -> **string**	concatenates two strings

5.2.2 Printing

Name	Type	Effect
`io_print`	**string** -> **void**	prints a string to the standard output

6 Formal Typing Rules

A typing rule takes the following shape:

$$\frac{\text{Hypotheses}}{S, \cdots \vdash_{\mathsf{type}} \texttt{grammar spec}}\ \textsc{name}$$

Here, S represents a stack of symbol definitions: $S \in (\mathsf{id} \times \mathsf{type} \times \{c, m\})^n$ 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`), type to the set of types in a given program (related to the Parser symbol `Type`), and $\{c, m\}$ to the mutability of the object: c represents an immutable value (as declared by **let**), and m a mutable one (declared by **mut**).

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 Subtyping Rules

6.1.1 Trivial Rule

$$\frac{}{\vdash_{\mathsf{T}} \texttt{t} \preceq \texttt{t}}\ \textsc{SubTyTrivial}$$

6.1.2 References

$$\frac{\vdash_T \texttt{t1} \preceq \texttt{t2}}{\vdash_T \texttt{t1} \preceq \texttt{t2?}}\ \textsc{SubTyRefs},\quad \frac{\vdash_T \texttt{t1} \preceq \texttt{t2}}{\vdash_T \texttt{t1?} \preceq \texttt{t2?}}\ \textsc{SubTyRefs}$$

6.1.3 Arrays

$$\frac{\vdash_T \texttt{t1} \preceq \texttt{t2}}{\vdash_T \texttt{[t1]} \preceq \texttt{[t2]}}\ \textsc{SubTyFuns}$$

6.1.4 Functions

$$\frac{\vdash_T \texttt{u1} \preceq \texttt{t1},\ \ldots,\ \vdash_T \texttt{un} \preceq \texttt{tn}\quad \vdash_T \texttt{rt} \preceq \texttt{ru}}{\vdash_T \texttt{(t1,...,tn)->rt} \preceq \texttt{(u1,...,un)->ru}}\ \textsc{SubTyFuncs}$$

6.1.5 Subtypes and Supertypes

The functions subtys and suptys of $\mathsf{types} \to \mathcal{P}(\mathsf{types})$ compute the sub- and supertype set of the input type, respectively.

They are used – among others – in the ExpLitArr rule.

$$\mathsf{subtys} := \begin{cases} \textbf{int} & \mapsto \{\textbf{int}\} \\ \textbf{flt} & \mapsto \{\textbf{flt}\} \\ \textbf{char} & \mapsto \{\textbf{char}\} \\ \textbf{bool} & \mapsto \{\textbf{bool}\} \\ \textbf{string} & \mapsto \{\textbf{string}\} \\ \texttt{t?} & \mapsto \{\texttt{u}, \texttt{u?} \mid \texttt{u} \in \mathsf{subtys}(\texttt{t})\} \\ \texttt{[t]} & \mapsto \{\texttt{[u]} \mid \texttt{u} \in \mathsf{subtys}(\texttt{t})\} \\ \texttt{(t1...tn)->rt} & \mapsto \{\texttt{(u1...un)->st} \mid \texttt{u1...un} \in \mathsf{suptys}(\texttt{t1...tn}), \texttt{st} \in \mathsf{subtys}(\texttt{rt})\} \end{cases}$$

$$\mathsf{suptys} := \begin{cases} \textbf{int} & \mapsto \{\textbf{int}\} \\ \textbf{flt} & \mapsto \{\textbf{flt}\} \\ \textbf{char} & \mapsto \{\textbf{char}\} \\ \textbf{bool} & \mapsto \{\textbf{bool}\} \\ \textbf{string} & \mapsto \{\textbf{string}, \textbf{string}?\} \\ \texttt{t?} & \mapsto \{\texttt{u?} \mid \texttt{u} \in \mathsf{suptys}(\texttt{t})\} \\ \texttt{[t]} & \mapsto \{\texttt{[u]}, \texttt{[u]?} \mid \texttt{u} \in \mathsf{suptys}(\texttt{t})\} \\ \texttt{(t1...tn)->rt} & \mapsto \left[\begin{array}{l} \{\texttt{(u1...un)->st}, \texttt{((u1...un)->st)?} \\ \quad \mid \texttt{u1...un} \in \mathsf{subtys}(\texttt{t1...tn}), \texttt{st} \in \mathsf{suptys}(\texttt{rt})\} \end{array}\right. \end{cases}$$

6.2 Builtin Operators

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

6.2.1 Unary Operators

6.2.1.1 Arithmetic Negation

$$\frac{}{\vdash_O \texttt{-} :: \textbf{int->int}}\ \textsc{TyUopNegInt},\quad \frac{}{\vdash_O \texttt{-} :: \textbf{flt->flt}}\ \textsc{TyUopNegInt}$$

6.2.1.2 Logical Negation

$$\frac{}{\vdash_O \texttt{!} :: \textbf{bool->bool}}\ \textsc{TyUopNot}$$

6.2.2 Binary Operators

6.2.2.1 Power

$$\frac{}{\vdash_O \texttt{**} :: \textbf{(int,int)->int}}\ \textsc{TyBopPowInt},\quad \frac{}{\vdash_O \texttt{**} :: \textbf{(flt,flt)->flt}}\ \textsc{tyBopPowFlt}$$

6.2.2.2 Multiplication

$$\frac{}{\vdash_O \texttt{*} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopMulInt}, \quad \frac{}{\vdash_O \texttt{*} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopMulInt}$$

6.2.2.3 Addition and Subtraction

- $$\frac{}{\vdash_O \texttt{+} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopAddInt}, \quad \frac{}{\vdash_O \texttt{+} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopAddFlt},$$
  $$\frac{}{\vdash_O \texttt{+} :: (\textbf{int},\textbf{char})\texttt{->}\textbf{char}}\ \textsc{TyBopAddCharR}, \quad \frac{}{\vdash_O \texttt{+} :: (\textbf{char},\textbf{int})\texttt{->}\textbf{char}}\ \textsc{TyBopAddCharL}$$
- $$\frac{}{\vdash_O \texttt{-} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopSubInt}, \quad \frac{}{\vdash_O \texttt{-} :: (\textbf{flt},\textbf{flt})\texttt{->}\textbf{flt}}\ \textsc{TyBopSubFlt},$$
  $$\frac{}{\vdash_O \texttt{-} :: (\textbf{int},\textbf{char})\texttt{->}\textbf{char}}\ \textsc{TyBopSubCharR}, \quad \frac{}{\vdash_O \texttt{-} :: (\textbf{char},\textbf{int})\texttt{->}\textbf{char}}\ \textsc{TyBopSubCharL}$$

6.2.2.4 Shift Operators

- $$\frac{}{\vdash_O \texttt{<<} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopLShift}$$
- $$\frac{}{\vdash_O \texttt{>>} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitRShift}$$
- $$\frac{}{\vdash_O \texttt{>>>} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitAShift}$$

6.2.2.5 Bitwise Operators

- $$\frac{}{\vdash_O \texttt{\&} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitAnd}$$
- $$\frac{}{\vdash_O \texttt{\^{}} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopXor}$$
- $$\frac{}{\vdash_O \texttt{|} :: (\textbf{int},\textbf{int})\texttt{->}\textbf{int}}\ \textsc{TyBopBitOr}$$

6.2.2.6 Logical Operators

- $$\frac{}{\vdash_O \texttt{\&\&} :: (\textbf{bool},\textbf{bool})\texttt{->}\textbf{bool}}\ \textsc{TyBopLogAnd}$$
- $$\frac{}{\vdash_O \texttt{\^{}\^{}} :: (\textbf{bool},\textbf{bool})\texttt{->}\textbf{bool}}\ \textsc{TyBopLogXor}$$
- $$\frac{}{\vdash_O \texttt{||} :: (\textbf{bool},\textbf{bool})\texttt{->}\textbf{bool}}\ \textsc{TyBopLogOr}$$

6.2.2.7 Comparison Operators

$$\frac{}{\vdash_{\mathsf{O}} \{\texttt{=}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\} :: \texttt{(}\textbf{\texttt{int,...,int}}\texttt{)->}\textbf{\texttt{bool}}} \ \textsc{TyCmpListInt},$$

$$\frac{}{\vdash_{\mathsf{O}} \{\texttt{=}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\} :: \texttt{(}\textbf{\texttt{flt,...,flt}}\texttt{)->}\textbf{\texttt{bool}}} \ \textsc{TyCmpListFlt},$$

$$\frac{}{\vdash_{\mathsf{O}} \{\texttt{=}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\} :: \texttt{(}\textbf{\texttt{char,...,char}}\texttt{)->}\textbf{\texttt{bool}}} \ \textsc{TyCmpListChar},$$

6.3 Expressions

6.3.1 Subtyping Expression Rule

To reduce tree rules, we define the following rules:

$$\frac{S \vdash_{\mathsf{E}} \texttt{exp} :: \texttt{t1} \quad \vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t}}{S \vdash_{\mathsf{ET}} \texttt{exp} \trianglelefteq \texttt{t}} \ \textsc{ExpSubLocal}, \quad \frac{S \vdash_{\mathsf{G}} \texttt{exp} :: \texttt{t1} \quad \vdash_{\mathsf{T}} \texttt{t1} \preceq \texttt{t}}{S \vdash_{\mathsf{GT}} \texttt{exp} \trianglelefteq \texttt{t}} \ \textsc{ExpSubGlobal}$$

6.3.2 Assignability

The rule $\vdash_{\mathsf{A}}$ defines when an expression can be assigned a value.

$$\frac{(\texttt{id}, \_, m) \in S}{S \vdash_{\mathsf{A}} \texttt{id}} \ \textsc{ExpAssnId}, \quad \frac{}{S \vdash_{\mathsf{A}} \texttt{e1[e2]}} \ \textsc{ExpAssnSub}$$

6.3.3 Global Expressions

The rule $\vdash_G$ describes global expressions, which are restricted in a way that they can be computed at compile time. Global variables are also strictly non-**`null`**.

The following copied rules use $\vdash_G$ instead of $\vdash_E$

6.3.3.1 Literals

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

6.3.3.2 Other Rules

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

Note that global variable declarations cannot feature function calls or **`null`** declarations.

6.3.4 Literals

- $$\frac{}{\vdash_{\mathsf{E}} n :: \textbf{\texttt{int}}} \ \textsc{ExpLitInt}$$
- $$\frac{}{\vdash_{\mathsf{E}} f :: \textbf{\texttt{flt}}} \ \textsc{ExpLitFlt}$$

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

- $$\frac{}{\vdash_{\mathsf{E}} \texttt{\textbf{true}} :: \texttt{\textbf{bool}}}\ \textsc{ExpLitBoolTrue},\quad \frac{}{\vdash_{\mathsf{E}} \texttt{\textbf{false}} :: \texttt{\textbf{bool}}}\ \textsc{ExpLitBoolFalse}$$

- $$\frac{}{\vdash_{\mathsf{E}} s :: \texttt{\textbf{string}}}\ \textsc{ExpLitString}$$

- In arrays, the typechecker looks for the common subtypes of all array literal elements and looks for the one type `t` that is a supertype of all elements and which is a subtype of all other such subtypes (the minimum of the subtypes given the $\preceq$ relation on types).

  $$\frac{S \vdash_{\mathsf{E}} \texttt{e1} :: \texttt{t1},\ \ldots,\ S \vdash_{\mathsf{E}} \texttt{en} :: \texttt{tn}\quad \texttt{t} = \min_{\preceq}(\bigcap_{\texttt{i}=1}^{\texttt{n}} \mathsf{suptys}(\texttt{ti}))}{S \vdash_{\mathsf{E}} \texttt{[e1,...,en]} :: \texttt{[t]}}\ \textsc{ExpLitArr}$$

  Because the graph connecting types and subtypes is a forest of trees, if the intersection of supertypes is nonempty there is a unique solution `t`.

6.3.5 Null

$$\frac{\texttt{t} \text{ is a non-null ref. type}}{S \vdash_{\mathsf{E}} \texttt{\textbf{null of}}\ \texttt{t} :: \texttt{t?}}\ \textsc{ExpNull}$$

6.3.6 Identifiers

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

6.3.7 Unary Operations

Unary and Binary Operations do not need to do subtype checking, as they operate only on primitives.

$$\frac{S \vdash_{\mathsf{E}} \texttt{exp} :: \texttt{t1}\quad \vdash_{\mathsf{O}} \texttt{op} :: \texttt{t1->t}}{S \vdash_{\mathsf{E}} \texttt{op exp} :: \texttt{t}}\ \textsc{ExpUop}$$

6.3.8 Binary Operations

$$\frac{S \vdash_{\mathsf{E}} \texttt{e1} :: \texttt{t1}\quad S \vdash_{\mathsf{ET}} \texttt{e2} :: \texttt{t2}\quad \vdash_{\mathsf{O}} \texttt{op} :: \texttt{(t1,t2)->t}}{S \vdash_{\mathsf{E}} \texttt{e1 op e2} :: \texttt{t}}\ \textsc{ExpBop}$$

6.3.9 Function Calls

$$\frac{S \vdash_{\mathsf{E}} \texttt{f} :: \texttt{(t1,...,tn)->rt}\quad \texttt{rt} \not\equiv \texttt{\textbf{void}}\quad S \vdash_{\mathsf{ET}} \texttt{e1} \trianglelefteq \texttt{t1},\ \ldots,\ S \vdash_{\mathsf{ET}} \texttt{en} \trianglelefteq \texttt{tn}}{S \vdash_{\mathsf{E}} \texttt{f(e1,...,en)} :: \texttt{rt}}\ \textsc{ExpFunc}$$

6.3.10 Subscript Access

- $$\frac{S \vdash_{\mathsf{E}} \texttt{s} :: \texttt{\textbf{string}}\quad S \vdash_{\mathsf{E}} \texttt{i} :: \texttt{\textbf{int}}}{S \vdash_{\mathsf{E}} \texttt{s[i]} :: \texttt{\textbf{char}}}\ \textsc{ExpArrSub}$$

- $$\frac{S \vdash_{\mathsf{E}} \texttt{e} :: \texttt{[t]}\quad S \vdash_{\mathsf{E}} \texttt{i} :: \texttt{\textbf{int}}}{S \vdash_{\mathsf{E}} \texttt{e[i]} :: \texttt{t}}\ \textsc{ExpArrSub}$$

6.3.11 Comparison Lists

$$\frac{\vdash_{\mathsf{O}} \texttt{op1} :: \texttt{(t0,t1)->t},\ \dots,\ \vdash_{\mathsf{O}} \texttt{opn} :: \texttt{(t(n-1),tn)->t} \quad S \vdash_{\mathsf{ET}} \texttt{e0} \trianglelefteq \texttt{t0},\ \dots,\ S \vdash_{\mathsf{ET}} \texttt{en} \trianglelefteq \texttt{tn}}{S \vdash_{\mathsf{E}} \texttt{e0 op1 ... opn en} :: \texttt{t}}\ \textsc{ExpCmpList}$$

6.4 Statements

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

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

6.4.1 Local Variable Declarations

$$\frac{(\texttt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{E}} \texttt{exp} :: \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \textbf{let}\ \texttt{id := exp} \Rightarrow (S \cup (\texttt{id}, \texttt{t}, c)), \bot}\ \textsc{StmtVDeclConst}$$

$$\frac{(\texttt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{ET}} \texttt{exp} \trianglelefteq \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \textbf{let}\ \texttt{id:t := exp} \Rightarrow (S \cup (\texttt{id}, \texttt{t}, c)), \bot}\ \textsc{StmtVTDeclConst}$$

$$\frac{(\texttt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{E}} \texttt{exp} :: \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \textbf{mut}\ \texttt{id := exp} \Rightarrow (S \cup (\texttt{id}, \texttt{t}, m)), \bot}\ \textsc{StmtVDeclMut}$$

$$\frac{(\texttt{id}, \_, \_) \notin_0 S \quad S \vdash_{\mathsf{ET}} \texttt{exp} \trianglelefteq \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \textbf{mut}\ \texttt{id:t := exp} \Rightarrow (S \cup (\texttt{id}, \texttt{t}, m)), \bot}\ \textsc{StmtVTDeclMut}$$

6.4.2 Assignments

$$\frac{S \vdash_{\mathsf{A}} \texttt{lhs} \quad S \vdash_{\mathsf{E}} \texttt{lhs} :: \texttt{t} \quad S \vdash_{\mathsf{ET}} \texttt{exp} \trianglelefteq \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \texttt{lhs := exp} \Rightarrow S, \bot}\ \textsc{StmtAssn}$$

6.4.3 Expression Statements

$$\frac{S \vdash_{\mathsf{E}} \texttt{exp} :: \texttt{t}}{S, \texttt{rt} \vdash_{\mathsf{S}} \texttt{exp} \Rightarrow S, \bot}\ \textsc{StmtExpr} \qquad \frac{S \vdash_{\mathsf{E}} \texttt{(t1,...,tn)->}\textbf{void}}{S, \texttt{rt} \vdash_{\mathsf{S}} \texttt{exp(a1,...,an)} \Rightarrow S, \bot}\ \textsc{StmtExprVoid}$$

6.4.4 If Statements

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

6.4.5 While Statements

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

6.4.6 Do-While Statements

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

6.4.7 Return Statements

$$\frac{S \vdash_{\mathsf{ET}} \texttt{exp} \trianglelefteq \texttt{rt}}{S, \texttt{rt} \vdash_{\mathsf{S}} \textbf{return}\ \texttt{exp} \Rightarrow S, \top}\ \textsc{StmtReturnExp}, \qquad \frac{}{S, \textbf{void} \vdash_{\mathsf{S}} \textbf{return} \Rightarrow S, \top}\ \textsc{StmtReturn}$$

6.4.8 Blocks

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

6.5 Global Statements

6.5.1 Global Function Declaration

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

6.5.2 Global Variable Declaration

$$\frac{(\mathsf{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{G}} \mathsf{exp} :: \mathsf{t} \quad \mathsf{t} \text{ non-}\mathbf{null}}{S \vdash_{\mathsf{G}} \mathbf{global}\ \mathtt{id\ :=\ exp} \Rightarrow S \cup \{(\mathsf{id}, \mathsf{t}, c)\}} \ \textsc{GStmtVDeclConst}$$

$$\frac{(\mathsf{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{GT}} \mathsf{exp} \trianglelefteq \mathsf{t} \quad \mathsf{t} \text{ non-}\mathbf{null}}{S \vdash_{\mathsf{G}} \mathbf{global}\ \mathtt{id{:}t\ :=\ exp} \Rightarrow S \cup \{(\mathsf{id}, \mathsf{t}, c)\}} \ \textsc{GStmtVDeclConst}$$

$$\frac{(\mathsf{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{G}} \mathsf{exp} :: \mathsf{t} \quad \mathsf{t} \text{ non-}\mathbf{null}}{S \vdash_{\mathsf{G}} \mathbf{global\ mut}\ \mathtt{id\ :=\ exp} \Rightarrow S \cup \{(\mathsf{id}, \mathsf{t}, m)\}} \ \textsc{GStmtVDeclMut}$$

$$\frac{(\mathsf{id}, \_, \_) \notin S \quad S \vdash_{\mathsf{GT}} \mathsf{exp} \trianglelefteq \mathsf{t} \quad \mathsf{t} \text{ non-}\mathbf{null}}{S \vdash_{\mathsf{G}} \mathbf{global\ mut}\ \mathtt{id{:}t\ :=\ exp} \Rightarrow S \cup \{(\mathsf{id}, \mathsf{t}, m)\}} \ \textsc{GStmtVDeclMut}$$

6.5.3 Program

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

6.6 Context Buildup

6.6.1 Global Function Declarations

$$\frac{(\mathsf{id}, \_) \notin S}{S \vdash_{\mathsf{GCF}} \mathbf{fn}\ \mathtt{id\ :\ a1{:}t1,\ ...,\ an{:}tn\ ->\ rt\ b} \Rightarrow S \cup (\mathsf{id}, \mathtt{(t1,...,tn)->rt}, c)} \ \textsc{GStmtFCtxtFDecl}$$

$$\frac{}{S \vdash_{\mathsf{GCF}} \mathbf{global}\ \mathtt{id\ :=\ exp} \Rightarrow S} \ \textsc{GStmtFCtxtVDeclConst}$$

$$\frac{}{S \vdash_{\mathsf{GCF}} \mathbf{global}\ \mathtt{id{:}t\ :=\ exp} \Rightarrow S} \ \textsc{GStmtFCtxtVTDeclConst}$$

$$\frac{}{S \vdash_{\mathsf{GCF}} \mathbf{global\ mut}\ \mathtt{id\ :=\ exp} \Rightarrow S} \ \textsc{GStmtFCtxtVDeclMut}$$

$$\frac{}{S \vdash_{\mathsf{GCF}} \mathbf{global\ mut}\ \mathtt{id{:}t\ :=\ exp} \Rightarrow S} \ \textsc{GStmtFCtxtVTDeclMut}$$

6.6.2 Program: Functions

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

6.7 Rule for Program Typechecking

Let $S^\star$ be the starting context which contains the builtin context. It looks as follows:

$$S^\star := \{\} \sqcup \left\{ \begin{array}{lll} (\mathtt{string\_of\_int}, & \mathbf{int}\ \mathtt{->}\ \mathbf{string}, & c), \\ (\mathtt{string\_of\_flt}, & \mathbf{flt}\ \mathtt{->}\ \mathbf{string}, & c), \\ (\mathtt{string\_concat}, & (\mathbf{string},\mathbf{string})\ \mathtt{->}\ \mathbf{string}, & c), \\ (\mathtt{io\_print}, & \mathbf{string}\ \mathtt{->}\ \mathbf{void}, & c) \end{array} \right\}$$

$$\frac{S^\star \vdash_{\mathsf{GC}} \mathsf{prog} \Rightarrow S' \quad S' \vdash_{\mathsf{G}} \mathsf{prog} \Rightarrow S}{\vdash \mathsf{prog}} \ \textsc{Prog}$$

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/
LiteralStr   ::= /"([^"\\]|(\\[\\nrt"]))*"/

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

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

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

null         ::= /null/
of           ::= /of/

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       ::= /\(/
RParen       ::= /\)/
LBrack       ::= /\[/
RBrack       ::= /\]/

QuestionMark ::= /\?/
```

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
                 | %Global      %Identifier %Colon Type %Assign GlobalExpression
                 | %Global %Mut %Identifier             %Assign GlobalExpression
                 | %Global %Mut %Identifier %Colon Type %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
                 | %Let %Identifier %Colon Type %Assign Expression
                 | %Mut %Identifier             %Assign Expression
                 | %Mut %Identifier %Colon Type %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
                  | RefType
                  | RefType %QuestionMark

RefType           ::=
                  | %Str
                  | %LBrack Type %RBrack

GlobalExpression ::= Expression*

Expression        ::=
                              | %Null %Of RefType
                              | %LBrack %RBrack %Of Type
                  | 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 ::=
                  | BaseExpression Application
                  | BaseExpression

BaseExpression    ::=
                  | %LParen Expression %RParen
                  | %LBrack CommaExpList %RBrack
                  | %LiteralInt
                  | %LiteralFlt
                  | %LiteralChar
                  | %LiteralBool
                  | %LiteralStr
                  | %Identifier

Application       ::=
                  | %LParen CommaExpListNE %RParen
                  | %LBrack Expression %RBrack

CommaExpList      ::=
                  | ϵ
                  | CommaExpListNE
CommaExpListNE    ::=
                  | Expression
                  | Expression %Comma CommaExpListNE
```