Oliver Graf

The Dromedar Programming Language

Contents

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
-	Unary Negation	100	non-assoc.	**int** -> **int** **flt** -> **flt**
!	Logical Negation			**bool** -> **bool**
**	Exponentiation	90	right	**int**,**int** -> **int** **flt**,**flt** -> **flt**
*	Multiplication	80	left	**int**,**int** -> **int** **flt**,**flt** -> **flt**
+	Addition	70	left	**int**,**int** -> **int** **flt**,**flt** -> **flt** **char**,**int** -> **char** **int**,**char** -> **char**
-	Subtraction			**int**,**int** -> **int** **flt**,**flt** -> **flt** **char**,**int** -> **char** **int**,**char** -> **char**
&	Bitwise And	60	left	**int**,**int** -> **int**
^	Bitwise Xor	50	left	**int**,**int** -> **int**
|	Bitwise Or	40	left	**int**,**int** -> **int**

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.

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.

6 Formal Grammar

6.1 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/

return      ::= /return/

Dash        ::= /\-/
Bang        ::= /!/
Star        ::= /\*/
Plus        ::= /\+/
LShift      ::= /<</
RShift      ::= />>/
AShift      ::= />>>/
Bitand      ::= /&/
Xor         ::= /\^/
Bitor       ::= /\|/

Assign      ::= /:=/

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

Comma       ::= /\,/

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

6.2 Parser Grammar

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

```
Program          ::=
                 | ϵ
                 | GlobalStatement Program

GlobalStatement  ::=
                 | GVDeclaration
                 | GFDeclaration

GVDeclaration    ::=
                 | %Global      %Identifier             %Assign GlobalExpression
                 | %Global %Mut %Identifier             %Assign GlobalExpression
                 | %Global      %Identifier %Colon Type %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
                  | ReturnStmt

VDeclaration      ::=
                  | %Let      %Identifier             %Assign Expression
                  | %Let %Mut %Identifier             %Assign Expression
                  | %Let      %Identifier %Colon Type %Assign Expression
                  | %Let %Mut %Identifier %Colon Type %Assign Expression

AssignStmt        ::=
                  | LHS %Assign Expression

IfStmt            ::=
                  | %If Expression Block ElifStmt

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

ReturnStmt        ::=
                  | %Return
                  | %Return Expression

LHS               ::=
                  | %Identifier

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

GlobalExpression ::= Expression*

Expression        ::=
                  | ExprPrec30
ExprPrec30        ::=
                  | ExprPrec40
                  | ExprPrec40 %Bitor ExprPrec40
ExprPrec40        ::=
                  | ExprPrec50
                  | ExprPrec50 %Xor ExprPrec60
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
```