Damo Language Reference Manual

Ian Covert (icc2115)
Hari Devaraj (sd2920)
Alan Gou (ajg2233)
Abhiroop Gangopadhyay (ag3661)

1. Identifiers

1.1 Data Types

Identifiers are how we assign names to variables, constants, and other data structures in Damo.

Identifiers can be of arbitrary length, but must always have their type specified during declaration. You do not need to assign a value upon declaration.

```
<type> <identifier> [= [value]]
int a;
int b = 10;
```

They can consist of any sequence of numbers and letters, in addition to underscores; however, every identifier must begin with a letter.

```
int valid_identifier = 10;
num 0_invalid_identifier = 7.5;
symbol _another_invalid_identifier;
```

Every identifier uniquely identifies a resource within its scope. In a given scope, there can only be one identifier with a given name. For example, the following set of statements is invalid:

```
int id = 10;
num id = 10.0;
def id(): void {
      /* function body */
}
```

1.2 Data structures

Using the <type>[<optional: initialization_count>] declaration means that we are declaring an array of a certain data type.

```
<type>[<initialization_count>] <identifier> [= [literal or
value]]
```

1.3 Keywords

There are certain reserved keywords in our language:

Built-in Types:

- int
- num
- symbol
- true
- false
- bool

Control Flow

- if
- else
- else if
- for
- while
- break
- continue

Functions

- def
- return
- void - default return value of functions

1.4 Dot Notation

For **symbols**, we can access their properties using dot notation.

```
// <symbol-identifier>.<property>
x.right; // accesses the right node property
```

1.5 Literals

We can declare various literals.

ints

```
    int literal_int = 153;
```

nums

```
    num literal_num = 17.5;
```

booleans

```
    bool false_bool = false;
    bool true_bool = true;
```

arrays

```
    int[5] literal_array = [1, 2, 3, 4, 5];
    int[5] another_literal;  // array of 5 zeroes
```

strings

```
    string literal_string = "Hello world";
```

symbol

```
    symbol z;
    symbol x;
    symbol y;
```

1.6 Comments

Damo has single-line comments. They are denoted by // and continue for the rest of the line.

```
    // This is a comment
    int a = 3;
    // a = 3 + 5;
    print(a) // prints 3, not 8
```

We have multi-line comments. Begin a multi-line comment with /* and end it with */. They cannot be nested.

```
    /*
    def non_used_function(): void {
         /* this is not an actual comment because we don’t have
    nested comments */
```

```
}
 */
```

1.7 Whitespace

Whitespace is not significant as a demarcator of function scope or any such thing in Damo. Thus, these two are equivalent:

```
def func() : void { if (3 == 3) { print("okay"); } else {
print("nope"); }}

def func() : void {
  if (3 == 3) {
    print("okay"); }
   } else {
    print("nope");
  }
}
```

1.8 End of lines

There are three kinds of syntactical structures that must end with a semicolon:

- Data type declarations
- Identifier assignments
- Function calls
- Single expressions;

```
int c = 15;
num numerical_value;
foo();
4 + 5;
```

There are three kinds of syntactical structures that do not end with a semicolon - these are statements that are bounded by curly braces

- if-else statements
- while/for loops
- function declarations

```
for (int i = 0; i < 10; i = i + 1) {
      print(i);
}

if (x > 5) {
      print("Greater!");
} else {
     print(“Not greater!”);
}

def foo() : void {
     /* function body */
}
```

2. Data Types:

2.1 Primitives:

Primitive data types act much like primitives in C. They can be declared and initialized later or initialized when declared, and passed by value.

Type	Description	Example

int	4 byte signed integer.	`int x;`
		`x = 1;`

num	8 byte floating point number.	`num y = 3;`
bool	1 byte boolean, is either true or false. True and false are always lowercase.	`bool = true;`
string	Basic way of expressing natural language. Language does not have C-like characters.	`string z = “what is my purpose?”;`

2.2 Composite Data Types:

These are always passed by reference.

Type	Description	Example

array	Can hold multiple instances of the same type. Size must be declared at time of initialization. Arrays are declared with original type followed by	`int a[2];`
		`a[0] = 1;`
		`a[1] = 2;`

	bracket notation.	

symbol	Basic building block for Mathematical functions. Has four members: operator, value, coefficient element flag, left node and right node. Can be implicitly instantiated by creating a mathematical expression with other symbols. When instantiated with an expression, the nodes form a dependency graph to represent a mathematical expression.	`symbol a;`
		`symbol b;`
		`symbol c;`
		`a = b + c;`

2.3 Symbols

If a math variable is instantiated with an expression with multiple operators, then the expression will be translated into it's fully parenthesized form according to order of operations, and the top level operation is assigned to the node to the left of the assignment operator. The user can use parentheses to specify which parts of the expression should be evaluation.

The Shunting-yard algorithm will be used to parse an expression. For every operation after the first operator, an additional implicit symbol node will be created to represent the value of the operator. If an equation is instantiated with a coefficient, then a symbol node will be created with the coefficient flag set to true and the numerical value will be placed in the node's value member. This allows users to traverse a function graph and avoid mutating nodes that are meant to be coefficients.

Example 1:
```
    symbol a;
    symbol b;
    symbol c;
    a = b + c;
    // the node a has b as a left pointer and c as a right pointer,
    with "+" as its operator
    // b and c have a as their parent and all other fields are
    empty
```

Example 2:
```
    symbol a;
    symbol b;
```

```
symbol c;
symbol d;
a = b + c * d;
/* Under the hood the expression is converted to the following:
a = b + ( c * d )
The parent node a has operator "+" and left pointer b and right
pointer to a node created under the hood representing the
parenthesized expression with a "*" as its operator and c as
its left pointer and d as its right pointer.
*/
// the implicitly created node can be accessed as follows:
symbol e;
e = a.right;
```

Example 3:

```
symbol a;
symbol b;
symbol c;
a = 2( b + c)
/* Under the hood, we have a 5 node dependency graph with the
node representing a being the root, and a.left.coefficient ==
true, because the node representing the 2 is a coefficient
*/
```

Example 4:

```
symbol x;
num y;
num z;
y = 2;
z = 3
x = y + z;
/* the x node has a coefficient flag set, and has value set to
5. z and y are both still nums*/
```

2.3 Arrays

You cannot assign arrays to other identifiers. Array sizes must be defined from instantiation in the following manner:

```
symbol a[10];
int[4] arr = [1, 2, 3, 4];
arr[0] = 2;
```

The individual indexes can be reassigned with individual members from the class.

3. Expressions and Operators

3.1 Expressions

Expressions are sequences of operators and operands. To see which sequences of operators and operands are valid, refer to the subsequent parts of this section. The following are example expressions:

```
42;                     // 42
3 ^ 2 + 4 ^ 2;          // 25
true or false;          // true
```

In examples such as the last one, the order in which expressions are evaluated depends on the precedence of the operators. For additional clarity, the programmer can use parentheses to group expressions:

```
(3 ^ 2) + (4 ^ 2);      // 25
(2 ^ (2 + 4)) ^ 2;      // 531,441
```

3.2 Assignment Operator

The single equals sign (=) is the assignment operator in Damo, and allows values to be associated with identifiers. The value to the left of the assignment operator must be an identifier (not an expression) and the value on the right must be an expression. The following are examples of assignments:

```
int x = 2;
num y = 7.0 / 2.5;
```

There are no shorthand operators for assignments of the following form:

```
x = x + 1;
y = y - 5;
```

3.3 Mathematical Operators

Mathematical operators can be used to create expressions combining ints and nums. The following table shows an exhaustive list of Damo's binary mathematical operators, in order of their precedence:

Operator	Description	Examples
_	Logarithm. The base is the left value and the argument is the right value.	`2 _ 4; // 2`
^	Exponentiation. The base is the left value and the exponent is the right value.	`4 ^ 2; // 16`
*	Multiplication.	`3 * 2; // 6`
/	Division.	`21 / 7; // 3`
+	Addition.	`1 + 1; // 2`
-	Subtraction.	`1 - 3; // -2`
%	Modulation.	`5 % 2; // 1`

The exact behavior of these operators depends on the types of the operands which are being combined. An operation involving two ints will return an int; an operation involving two nums will return a num; and an operation involving one int and one num will return a num. In order to have an expression involving two ints (such as 7 / 2) evaluate to a num, one or more of the operands must be converted to a num.

3.4 Comparison Operators

Comparison operators can be used to create expressions resulting in boolean values. The following table shows an exhaustive list of Damo's comparison operators:

Operator	Description	Examples

<	Less than.	`1 < 1; // false`
		`1 < 2; // true`

<=	Less than or equal to.	`1 <= 1; // true`

>	Greater than.	`1 > 1; // false`
		`2 > 1; // true`

>=	Greater than or equal to.	`1 >= 1; // true`
==	Equal to.	`1 == 1; // true`

		`1 == 2; // false`
		`“Hi” == “Hi”; // true`

!=	Not equal to.	`1 != 1; // false`
		`1 != 2; // true`

Comparison operators can be used to combine operands of compatible types. For example, ints can be compared with ints or nums; strings can be compared with strings.

3.5 Boolean Operators

Boolean operators are used to create expressions where the operands are boolean values. The following table contains a list of Damo’s boolean operators, in order of their precedence:

Operator	Description	Examples
not	Logical not (unary).	`not true; // false`
and	Logical and.	`true and true; // true`
or	Logical or.	`true or false; // true`

3.6 Typecasts

Type casts can be used to cause the result of an expression to become a different data type. Type casts only work between compatible types. The following are examples:

```
num x = 2.5;
int y = int(x);  // 2
```

The table below shows an exhaustive list of which data types can be casted to which other types:

Data Type	Can be Casted to	Examples

int	num, string	`num x = num(1);`
		`string s = string(1); //`
		`”1”`

num	int, string. Casting a num to an int causes the value to be truncated.	`int x = int(2.5);`

bool	int, num, string. Casting a boolean to an int of num results in either 0 or 1.	`int x = int(true); // 1`
		`num y = num(false); // 0.0`

3.7 Array Subscripts

Elements of arrays can be accessed using square brackets. Arrays are zero-index.

When on the left side of an assignment operator, the specified element of the array is mutated. Otherwise, when used in an expression, the element at the specified position is returned.

The following are examples:

```
int[4] arr = [1, 2, 3, 4];
arr[0] = 2;
print(arr[0]); // prints 2
```

3.8 Function Calls as Expressions

Function calls can be used in expressions, as long as the function returns a value. The following is an example of an expression containing a function:

```
def trivial int () : int {
    return 1;
}
trivial() + 2; // evaluates to 3
```

4. Statements

An expression is formed from a combination of operations and operands. A statement can be an assignment, a function call, or any other expression, including control flow statements (such as if-else, while loops, and for loops). Every statement must end with a semicolon, aside from control flow statements. Any expression that ends with a semicolon becomes a statement. Examples include the following:

```
int a = 5;
```

```
      num b = 5.2;
```

Groups of statements and declarations can be grouped together in curly braces, and will be treated as an expression. It is important to note that in this case, no semicolon is needed after the curly brace. We use curly braces in if-else statements, while statements, and for statements, which we discuss below. Note that every expression in the parentheses below in our discussion of control flow elements (if-else, while, for) must be of type bool (for example, if (int a = 5) is not a valid expression, but if (int a == 5) is fine).

4.1 If-Else Statements

We allow for conditional statements with branches in the form of an if-elif-else statement. Formally, the syntax is as follows:

```
      if (expression){
            statement;
      }
      elif (expression){
            statement;
      }
      else{
            statement;
      }
```

4.2 While Loops

We also allow while loops. The syntax is as follows:

```
      while (expression){
            statement;
      }
```

The statement in the body of the loop executes until the expression in the parentheses is no longer true.

4.3 For Loops

Another kind of loop supported by the language is the for loop. The syntax is as follows:

```
      for(expression; expression; expression){
        statement;
    }
```

The first statement is the initializer. It sets the initial state of the loop. The second expression is the boolean check. This is analogous to the expression in the parentheses of the while loop seen earlier, and ensures that the loop's current state still satisfy the conditions of the loop. The third statement in the parentheses updates the state of the loop. Below is an example:

```
    int y = 4;
    for(int x=5; x<6; x=x+1){
        y=y+1;
    }
```

The final value of y after this loop runs is 5.
Note that we declared a new variable in the for loop. We could have also declared it outside the for loop and either instantiated it or reassigned it to a new value in the loop. For example, we could have had the following:

```
    int y = 4;
    int x;
    for(x=5; x<6; x=x+1){
        y=y+1;
    }
```

4.4 Break

The break statement is used to exit from a loop before the loop condition is violated. Break statements can be used in both while and for loops. Below is an example:

```
    int y = 4;
    for(int x=5; x<10; x=x+1){
        y=y+1;
        if(y==6){
            break;
        }
    } //this loop runs twice, and then breaks, even though x is
    still // less than 10
```

Note that break statements break only out of the innermost loop.

4.5 Continue

The continue statement causes the next iteration of the enclosing loop. This means that in the loop, the control flow will go back to the top of the loop and the loop condition will be evaluated again. Below is an example:

```
int y = 4;
for(int x=5; x<10; x=x+1){
     if(x==6){
          continue;
     }
     y = y + 1;
}
// This doesn’t increment y if x = 6, but control flow returns
to // the top of the loop to recheck the condition
```

5. Functions

5.1 User-defined functions:

User-defined functions are declared in the following format:

```
def function_name(arg_type arg_name, arg_type arg_name, …) :
return_type {
     ….
}
```

Below is an example function that returns:

```
def increment (int a) : int {
     a = a+1;
     return a;
}
```

Below is an example function that does not return:

```
def print_string(string s) : void {
     print(s);
}
```

5.2 Calling Functions:

Functions must be declared before they can be called. Functions can be called just by invoking the function name and supplying any parameters.

```
<function-name>(<parameter>)
```

We can also assign variables to the return types of functions, though only if the function has a non-void return type:

```
data_type var_name = function(<parameter>);
```

We can extend the increment example above.

```
int x = 5;
x = increment(5); // x contains the value 6
```

5.3 Standard Library Functions

The standard library contains a suite of functions dealing with symbolic representations of functions, the nodes, and any manipulations of such nodes. These include canCompute (which checks to see if a computation over some relationship is valid, such as taking the derivative with respect to a variable not in the relationship), grad (compute gradient) and eval (evaluate function) among others.

6. Program Structure and Scope

6.1 Scope

All functions, data types, and primitives defined in a file are inside the **scope** of the file. We use lexical scoping.

Every file has its own scope.

Variables declared in the file are accessible inside functions that are declared in the same scope.

mainFile.damo

```
int int_value = 5;
int two = 2;

def multiply_by_two(int number) : int {
  return number * two;
}

print(multiply_by_two(int_value)) // prints '10'
```

6.2 Creation of Scope

Scope is created by three things:

- if/else statements
- while/for loops
- function declarations

Variables declared inside one of these three scope-creating statements leave their containing scopes when the statement ends.