Delta Language Specification

Emil Laine
emil@cs.helsinki.fi

September 15, 2018

This document describes the syntax and semantics of the Delta programming language.

Note: This document is incomplete.

Contents

Chapter 1

Lexical structure

1.1 Keywords

The following keywords are reserved and can't be used as identifiers.

`addressof`	`false`	`sizeof`
`break`	`for`	`static`
`case`	`goto`	`struct`
`catch`	`if`	`switch`
`const`	`import`	`this`
`continue`	`in`	`throw`
`def`	`init`	`throws`
`default`	`inline`	`true`
`defer`	`interface`	`try`
`deinit`	`mutable`	`typealias`
`do`	`mutating`	`undefined`
`else`	`null`	`var`
`enum`	`private`	`while`
`extern`	`public`	`_`
`fallthrough`	`return`	

1.2 Operators and delimiters

Binary arithmetic operators:

<table>
<tr><td><code>+</code></td><td><code>/</code></td><td><code>&</code></td><td><code>||</code></td></tr>
<tr><td><code>-</code></td><td><code>**</code></td><td><code>&&</code></td><td><code>^</code></td></tr>
<tr><td><code>*</code></td><td><code>%</code></td><td><code>|</code></td><td><code><<</code></td></tr>
</table>

```
>>      /=      &&=     <<=
+=      **=     |=      >>=
-=      %=      ||=
*=      &=      ^=
```

Binary comparison operators:

```
==      <       <=
!=      >       >=
```

Miscellaneous binary operators:

```
=       ..      ...
```

Unary prefix operators:

```
+       *       !
-       &       ~
```

Unary postfix operators:

```
++      --      !
```

Delimiters:

```
(       )       .       ;
[       ]       ,
{       }       :       ->
```

From the above sets of operators, the following are overloadable by user code: `+` (both unary and binary), `-` (both unary and binary), `*`, `/`, `%`, `==`, `<`.

1.3 Comments

Delta has two kinds of comments:

- Line comments that start with `//` and continue until the end of the line.
- Block comments that start with `/*` and end with `*/`. Block comments can be nested.

1.4 Literals

1.4.1 Integer literal

$$\begin{aligned}
&\textit{binary-digit} \to \texttt{0} \mid \texttt{1}\\
&\textit{octal-digit} \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7}\\
&\textit{decimal-digit} \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}\\
&\textit{nonzero-decimal-digit} \to \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}\\
&\textit{lowercase-hex-digit} \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f}\\
&\textit{uppercase-hex-digit} \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F}
\end{aligned}$$

$$\begin{aligned}
&\textit{binary-integer-literal} \to \texttt{0b}\textit{binary-digit}+\\
&\textit{octal-integer-literal} \to \texttt{0o}\textit{octal-digit}+\\
&\textit{decimal-integer-literal} \to \textit{nonzero-decimal-digit}\ \textit{decimal-digit}* \mid \texttt{0}\\
&\textit{hex-integer-literal} \to \texttt{0x}(\textit{lowercase-hex-digit}+ \mid \textit{uppercase-hex-digit}+)
\end{aligned}$$

$$\begin{aligned}
&\textit{integer-literal} \to \textit{binary-integer-literal}\\
&\textit{integer-literal} \to \textit{octal-integer-literal}\\
&\textit{integer-literal} \to \textit{decimal-integer-literal}\\
&\textit{integer-literal} \to \textit{hex-integer-literal}
\end{aligned}$$

1.4.2 Floating-point literal

Floating-point literals have the following form:

$$\begin{aligned}
&\textit{floating-point-literal} \to \textit{nonzero-decimal-digit}\ \textit{decimal-digit}*\texttt{.}\,\textit{decimal-digit}+\\
&\textit{floating-point-literal} \to \texttt{0.}\,\textit{decimal-digit}+
\end{aligned}$$

1.4.3 Boolean literal

$$\textit{boolean-literal} \to \texttt{true} \mid \texttt{false}$$

1.4.4 Null literal

$$\textit{null-literal} \to \texttt{null}$$

1.4.5 String literal

$$\begin{aligned}
&\textit{string-literal} \to \texttt{"}(\textit{character} \mid \textit{interpolated-expression})*\texttt{"}\\
&\textit{interpolated-expression} \to \texttt{\$\{}\ \textit{expression}\ \texttt{\}}
\end{aligned}$$

1.4.6 Array literal

$$\textit{array-literal} \to \texttt{[}\ \textit{elements}\ \texttt{]}$$

where *elements* is a comma-separated list of zero or more expressions of the same type.

1.4.7 Tuple literal

> *tuple-literal* → `(` *elements* `)`

where *elements* is a comma-separated list of zero or more *tuple-literal-elements*:

> *tuple-literal-element* → *identifier* `:` *expression*
> *tuple-literal-element* → *identifier*

The second form is a shorthand for *tuple-literal-elements* of the form '*identifier* `:` *identifier*'.

1.5 Identifiers

> *identifier-first-character* → upper- or lowercase letter `A` through `Z`
> *identifier-first-character* → `_`
>
> *identifier-character* → *identifier-first-character*
> *identifier-character* → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
>
> *identifier* → *identifier-first-character* *identifier-character**

Chapter 2

Types

type $\rightarrow$ *basic-type*
type $\rightarrow$ *pointer-type*
type $\rightarrow$ *optional-type*
type $\rightarrow$ *array-type*
type $\rightarrow$ *function-type*
type $\rightarrow$ *tuple-type*
type $\rightarrow$ `mutable` *basic-type*
type $\rightarrow$ `mutable` *function-type*
type $\rightarrow$ `mutable` *tuple-type*

2.1 Basic types

basic-type $\rightarrow$ *identifier*
basic-type $\rightarrow$ *identifier* < *generic-argument-list* >
generic-argument-list $\rightarrow$ comma-separated list of one or more *types*

2.1.1 Integer types

There are eight fixed-width integer types: `int8`, `int16`, `int32`, `int64`, and their unsigned counterparts `uint8`, `uint16`, `uint32`, `uint64`. The language also provides:

- `byte` and `ubyte`, which have at least 8 bits
- `short` and `ushort`, which have at least 16 bits
- `int` and `uint`, which have at least 32 bits
- `long` and `ulong`, which have at least 64 bits

Overflow is undefined for all integer types, both signed and unsigned, to aid optimization. Overflow checks are enabled by default, and can be disabled with a compiler option, or by compiling in unchecked mode. The standard library provides arithmetic functions that have defined behavior on overflow.

2.1.2 Floating-point types

There are three fixed-width floating-point types: `float32`, `float64`, and `float80`. The language also provides:

- `float`, which has at least 32 bits
- `double`, which has at least 64 bits

2.1.3 Struct types

Structs are composite data types which can be defined using the `struct` keyword.

2.1.4 Interface types

The `interface` keyword declares an interface, i.e. a set of requirements (member functions and properties). Types that are declared to implement an interface `I` and fulfill `I`'s requirements can be used as values for a variable of type `I`. This enables runtime polymorphism. Like structs, interfaces may be generic.

2.2 Pointer types

Pointers are values that point to other values. They can be reassigned to point to another value (if the pointer type itself is declared as `mutable`), but they must always refer to some value, i.e. they cannot be null by default (nullable pointers can be created using the optional type, see below). Member access, member function calls, and subscript operations via pointers are allowed: they will be forwarded to the pointee value.

pointer-type → *pointee-type* `*`
pointer-type → *pointee-type* `mutable *`
pointee-type → *type*

The *pointee-type* may be mutable. Prefixing the `*` with `mutable` makes the *pointer-type* itself mutable.

Pointer arithmetic is supported in the form of the following operations:

- *pointer* `+` *integer*
- *pointer* `+=` *integer*
- *pointer* `++`
- *pointer* `-` *integer*
- *pointer* `-=` *integer*
- *pointer* `--`
- *pointer* `-` *pointer*

2.3 Array types

array-type-with-constant-size → *element-type* `[` *size* `]`
array-type-with-runtime-size → *element-type* `[ ]`
array-type-with-unknown-size → *element-type* `[ ? ]`
element-type → *type*

array-type-with-constant-size represents a contiguous block of *size* elements of type *element-type*. *array-type-with-runtime-size* is conceptually a pointer-and-size pair. *array-type-with-unknown-size* can only be used through a pointer; such pointers are memory-layout-compatible with pointers to *element-type*, primarily for C interoperability. *array-type-with-unknown-size* is the only array type for which index operations are not guaranteed to be bounds-checked.

The *element-type* may be mutable.

2.4 Optional type

> "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."
>
> — C. A. R. Hoare

An object of the optional type `T?` (where `T` is an arbitrary type) may contain a value of type `T` or the value `null`.

optional-type → *wrapped-type* `?`
wrapped-type → *type*

2.5 Function types

Function types are written out as follows:

function-type → `(` *parameter-type-list* `)` `->` *return-type*
parameter-type-list → comma-separated list of zero or more *types*
return-type → *type*

The types in the *parameter-type-list* and *return-type* may not have a top-level mutable modifier.

2.6 Tuple types

tuple-type → `(` *tuple-element-list* `)`
tuple-element-list → comma-separated list of one or more *tuple-elements*
tuple-element → *name* `:` *type*

Tuples behave like structs, but they're defined inline. Tuples are intended as a lightweight alternative for situations where defining a whole new struct feels overkill or inappropriate, e.g. returning multiple values from a function.

While struct types are considered the same only if they have the same name, tuple types are considered the same if their structure is the same, i.e. if they have the same number of elements in the same order with the same names and types.

2.6.1 Tuple unpacking

The elements of a tuple value may be unpacked into individual variables as follows:

> *tuple-unpack-statement* → *variable-list* `=` *tuple-expression* `;`

variable-list is a comma-separated list of one or more variable names. The variable names must match the element names of the *tuple-expression*, and be in the same order.

Chapter 3

Declarations

3.1 Variables

Variable declarations introduce a new variable into the enclosing scope. The syntax is as follows:

> *implicitly-typed-variable-definition* → `var` *variable-name* `=` *initializer* `;`
> *explicitly-typed-variable-definition* → `var` *variable-name* `:` *type* `=` *initializer* `;`
> *variable-declaration* → `var` *variable-name* `:` *type* `;`

If *type* is present, the variable has the specified type. The compiler ensures that the given *initializer* is compatible with this type. If no *type* is given, the compiler will infer the type of the variable from the *initializer*. The *initializer* is an expression that provides the initial value for the variable.

If *type* has been specified, *initializer* may also be the keyword `undefined`, in which case the variable is not initialized and all use-before-initialization warnings for the variable will be suppressed. Reading from an uninitialized variable causes undefined behavior.

In *variable-declaration*, the variable is declared but not initialized. This allows delayed initialization, which causes the compiler to enforce that the variable is always initialized properly before its value is accessed.

3.2 Constants

Constant declarations introduce a named compile-time constant into the enclosing scope:

> *implicitly-typed-constant-definition* → `const` *constant-name* `=` *initializer* `;`
> *explicitly-typed-constant-definition* → `const` *constant-name* `:` *type* `=` *initializer* `;`

Constant declarations must always have an initializer. The compiler evaluates the initializer at compile time.

3.3 Functions

A function can be defined with either of the following syntaxes:

> `def` *function-name* `(` *parameter-list* `)` `{` *function-body* `}`
> `def` *function-name* `(` *parameter-list* `)` `:` *return-type* `{` *function-body* `}`

The return type of the first version is `void`. The *parameter-list* is a comma-separated list of *parameters*:

> *parameter* → *parameter-name* `:` *parameter-type*

parameter-name is an identifier specifying the name of the parameter. A function cannot have multiple parameters with the same name.

return-type defines what kind of values the function can return. This may be a tuple type to allow the function to return multiple values without having to define a whole new struct type.

Parameter and return types may not have a top-level `mutable` modifier.

A function declaration may optionally be prefixed with any number of *function-specifiers*.

3.3.1 Member functions

Member functions are just like normal functions, except that they receive an additional parameter, called the "receiver", on the left-hand-side of the function call, separated by a period:

> *member-function-call* → *receiver* `.` *member-function-name* `(` *argument-list* `)`

Member functions are defined with the same syntax as non-member functions, but are written inside a type declaration. That type declaration defines the member function's receiver type. Inside member functions, the receiver can be accessed with the keyword `this`.

3.3.1.1 Initializers

Initializers are a special kind of member functions that are used for initializing newly created objects.

> *initializer-definition* → `init` `(` *parameter-list* `)` `{` *body* `}`

Initializers can be invoked with the following syntax:

> *initializer-call* → *receiver-type* `(` *argument-list* `)`

The *initializer-call* expression returns a new instance of the specified type that has been initialized by calling the initializer function with a matching parameter list.

3.3.1.2 Deinitializers

Deinitializers are another special kind of member functions. They are automatically called on objects when they're destroyed, but can also be called explicitly. They can be used e.g. to deallocate resources allocated in an initializer. They are declared as follows:

> *deinitializer-definition* → `deinit` `(` `)` `{` *body* `}`

3.3.2 Private and public functions

Both member functions and global functions may be declared private or public by prefixing the function definition with the keyword `private` or `public`. Private functions are only accessible from the file they're declared in. Public functions are accessible from anywhere, including other modules. Functions not marked private or public are *module-private*, i.e. only accessible within the module they're declared in.

3.3.3 Function specifiers

function-specifier → `inline`

3.3.3.1 `inline`

A function defined with the `inline` keyword is an *inline function.* Inline functions are guaranteed to be inlined when compiling in debug mode (without optimizations). When compiling with optimizations, the compiler may choose to not inline an inline function.

3.4 Structs

Structs are defined as follows:

struct-definition → `struct` *struct-name* `{` *member-list* `}`

struct-name becomes the name of the struct. *member-list* is a sequence of *member-variable-declarations* and *member-function-declarations.* Structs can be declared to implement interfaces by listing the interfaces after a `:` following the struct name:

`struct` *struct-name* `:` *interface-list* `{` *member-list* `}`

The *interface-list* is a comma-separated list of one or more interface names. The compiler will emit an error if the struct doesn't fulfill all the requirements of a specified interface.

3.4.1 Member variables

Structs can contain member variables. The syntax of a member variable definition is as follows:

member-variable-declaration → `var` *member-variable-name* `:` *type* `;`

3.4.2 Generic structs

Generic structs can be declared as follows:

`struct` *struct-name* `<` *generic-parameter-list* `>` `{` *member-list* `}`

where *generic-parameter-list* is a comma separated list of one or more *generic-parameters*:

generic-parameter → *generic-type-parameter*
generic-type-parameter → *identifier*

The identifier of a *generic-type-parameter* serves as a placeholder for types used to instantiate the generic struct.

Chapter 4

Statements

4.1 Assignment statement

assignment-statement → *lvalue-expression* `=` *expression* `;`
assignment-statement → `_` `=` *expression* `;`

Assignments in Delta don't return any value. This applies to compound assignments as well, including `++` and `--` (see below). Furthermore, this obsoletes the two different forms of `++` and `--`, so only the postfix versions are valid as syntactic sugar for `+= 1` and `-= 1`, respectively.

The assignment to `_`, called the *discarding-assignment*, can be used to ignore the result of the right-hand side expression, suppressing any compilation errors or warnings that would otherwise be emitted.

4.2 Increment and decrement statements

increment-statement → *lvalue-expression* `++` `;`
decrement-statement → *lvalue-expression* `--` `;`

4.3 Block

block → `{` *statement** `}`

4.4 `if` statement

if-statement → `if` `(` *expression* `)` *block*
if-statement → `if` `(` *expression* `)` *block* `else` *block*
if-statement → `if` `(` *expression* `)` *block* `else` *if-statement*

4.5 `return` statement

return-statement → `return` $\textit{expression}_{\text{opt}}$ `;`

4.6 `for` statement

The `for` statement loops over a range. The syntax is as follows:

for-statement → `for ( var` *identifier* `in` *range-expression* `)` *block*

4.7 `while` statement

The `while` statement loops until a condition evaluates to `false`. The syntax is as follows:

while-statement → `while (` *condition* `)` *block*

4.8 `switch` statement

switch-statement → `switch (` *expression* `) {` *case+* `}`
case → `case` *expression* `:` *statement+*
case → `default :` *statement+*

In addition to integer types, the `switch` statement can be used to match strings.

The cases in a `switch` statement don't fall through by default. The fall-through behavior can be enabled for a individual cases with the `fallthrough` keyword.

`switch` statements must be exhaustive if *expression* is of an enum type. This is enforced by the compiler.

4.9 `defer` statement

The `defer` statement has the following syntax:

defer-statement → `defer` *block*

The *block* will be executed when leaving the scope where the *defer-statement* is located. Multiple deferred blocks are executed in the reverse of the order they were declared in. Return statements are disallowed inside the defer block.

Chapter 5

Expressions

5.1 Unary expressions

prefix-unary-expression → *operator operand*
postfix-unary-expression → *operand operator*

5.1.1 Unwrap expression

unwrap-expression → *operand* `!`

The *unwrap-expression* takes an operand of an optional type, and returns the value wrapped by the optional. If the operand is null, an assertion error will be triggered, except in unchecked mode, where the compiler may assume that the operand is never null.

5.2 Binary expression

binary-expression → *left-hand-side binary-operator right-hand-side*

5.3 Conditional expression

conditional-expression → *condition* `?` *then-expression* `:` *else-expression*

5.4 Member access expression

member-access-expression → *expression* `.` *identifier*

5.5 Subscript expression

subscript-expression → *expression* `[` *expression* `]`

5.6 Function call expression

call-expression → *expression* `(` *argument-list* `)` *argument-list* is a comma-separated list of zero or more *argument-specifiers*:

> *argument-specifier* → *unnamed-argument* | *named-argument*
> *unnamed-argument* → *expression*
> *named-argument* → *argument-name* `:` *expression*

argument-name is an identifier specifying the name of the parameter the argument *expression* is being assigned to.

5.7 Range expression

> *exclusive-range-expression* → *lower-bound* `..` *upper-bound*
> *inclusive-range-expression* → *lower-bound* `...` *upper-bound*

5.8 Closure expression

> *closure-expression* → `(` *parameter-list* `)` `->` *expression*
> *closure-expression* → `(` *parameter-list* `)` `->` *block*

Specifying the type for parameters in a closure *parameter-list* is optional. Omitting the type (and the corresponding colon) causes the type for that parameter to be inferred from the context.

If the closure *parameter-list* only contains one parameter, the enclosing parentheses may be omitted.

Chapter 6

Standard library

6.1 Types

6.1.1 String types

The type `String` holds sequences of Unicode characters.

6.1.2 Range types

The standard library defines the following two generic types to represent ranges:

- `Range<T>` for ranges with an exclusive upper bound
- `ClosedRange<T>` for ranges with an inclusive upper bound