Language Documentation

Ruben Saunders

January 2025

Contents

1 Introduction

This document will contain information about the Edel language. As an educational tool, it is designed to be simple and easy to learn and operate, with the goals of being clear a readable. Each section is written from the perspective of little programming knowledge.

1.1 Overview

As the most wide-spread paradigm, this language is imperative by nature. Its syntax is inspired by C, but borrows features from C++ such as namespaces and operator overloading. However, unlike these two languages, names are *forward declared*, meaning they can be used prior to their definition. This foregoes the need for header files of explicit forward declarations as in C and C++, and allows a much freer programming experience similar to modern languages.

1.2 The Compiler

A program consists of one file (the entry file), which is compiled by the compiler to produce an assembly output.

```
$ ./compiler <input_file> [-o <output_file>] [flags]
```

Where `[flags]` are as follows:

- `--ast` - prints the parsed program's AST.

Note that the output file is optional.

1.3 "Hello, World"

As is customary, a quick "hello world" program written in Edel.

```
func main() {
    print("Hello, world\n");
}
```

2 Basic Syntax

Inspires by C, Edel is not whitespace sensitive, with scopes marked by braces {}. Semicolons are not requires after every line, namely closing braces, but are required after expressions.
Comments come in two forms: single– and multi-line. Single-line comments `// ...` comment out the remainder of the current line; multi-line comments `/* ... */` comment out anything they enclose.

2.1 Variables

Using the common analogy, a variable is like a labeled box which stored data. Each box has a *type* which tells us what it stores. For example, "int 3". More technically, a variable is a location in memory where data can be stored, with its type determining the size of the variable and how its value can be used.
Variables are defined using the `let` keyword, with the variables type following a colon. For example, to define our "int 3" from earlier,

```
let foo: int = 3;
```

If required, multiple variables can be defined in the same `let` statement.

```
let foo: int = 3, bar: float = 3.14;
```

If the type is omitted, it is deduced from the assigned value. Hence, this can only be done if the variable is assigned.

```
let foo = 3, bar = 3.14;
```

(In this case, both variables adopt the types as previously written.)

2.1.1 Shadowing

Edel allows names to be *shadowed*, meaning a variable may re-use a variable's name.
In non-local shadowing, the previous definition becomes visible again after the current scope is exited.

```
let foo = 1;
{
    let foo = 2;
    // foo = 2
}
// foo = 1
```

Edel also allows local shadowing. In this case, once shadowed, the previous definition is essentially lost and cannot be referenced.

```
let foo = 1;
// foo = 1
let foo = 2;
// foo = 2
```

2.1.2 Constants

Constants are values that, once defined, cannot be changed. Constants may be defined using the `const` keyword in place of `let`.

```
const pi = 3.14159;
```

In the case of multiple definitions, *all* symbols will be defined as const.

2.2 Expressions & Statements

Generally, a statement is some code which does not produce a value, whereas an expression does. In Edel, *everything* is an expression, with statements, by default, returning the unit type '`()`'.
Expressions are expected to be terminated by a semicolon. The exception to this is the final expression in a block; if omitted, the block 'returns' the result of this expression.

```
let a = { 1 + 2; }; // a = ()
let b = { 1 + 2 }; // b = 3
let c = { let z = 2 * b; z } + 2; // c = 8
```

Note that only the last expression in a block may have their semicolon emitted. By always terminating expressions, the block will act like a statement.

2.2.1 Value Categories

Each expression, be it an operator with operands, literals, or variable names etc., is characterised by a *type* and a *category*. Types, covered in section 3, give meaning to data and what it means in context, whereas the category tells dictates *how* a value can be used.
Edel has two categories, be them l– and r-values. The former is a value which refers to a memory location, such as variables, while the latter has no identifiable location in memory. The *l* and *r* stand for *left* and *right* because the respective values may appear on the left– or right-hand side of an assignment operator. That is,

```
a = 1; // a (lvalue) = 1 (rvalue)
```

As an rvalue has no identifiable location, it cannot be assigned to, nor can it be access via the dot '`.`' operator. Instances of misuse of an l– or rvalue results in an error "`expected l/rvalue, got type`".

```
1 = 2; // error: expected lvalue, got i32
true.a; // error: expected lvalue, got bool

namespace maths {}
maths + 1; // error: expected rvalue, got namespace
```

2.3 Code Blocks

As a whitespace-ignorant language, Edel uses blocks to denote structure and scope. A block is introduced by braces {} and contains a section of code. A block may be written anywhere a statement is expected, allowing the programmer to instead write multiple statements in its body. Finally, in Edel, every block introduces a new lexical scope 7.

3 Types

Continuing the analogy from variables, a type is a label that tells the program what kind of data a variable holds. Every variable and value has a type; a value without a type is meaningless, as a type gives data a meaning.

3.1 Primitive Types

Primitive types are a subset of those provided by the compiler representing atomic values, such as numbers.

Type	Alias	Description
`bool`		A truth value – either `true` or `false`
`u8`	`byte`	Unsigned 8-bit integer
`i8`		Signed 8-bit integer
`u16`		Unsigned 16-bit integer
`i16`		Signed 16-bit integer
`u32`		Unsigned 32-bit integer
`i32`	`int`	Signed 32-bit integer
`u64`		Unsigned 64-bit integer
`i64`	`long`	Signed 64-bit integer
`f32`	`float`	32-bit floating-point number
`f64`	`double`	64-bit floating-point number

Table 1: Primitive Types

These types have the following type hierarchy:

- `int8` < `int16` < `int32` < `int64`
- `uint8` < `uint16` < `uint32` < `uint64`
- `uint`n < `int`$(n+k)$
- `float32` < `float64`
- `[u]int`n < `float64`
- `[u]int`n < `float32` where $n < 64$.

Note that `bool` is not related to anything. Unlike in other language, `bool` is treated as a distinct type with abstract true/false values, rather than a synonym for integers 1/0, respectively.

3.2 The Unit Type

Ths unit type, denotes by empty brackets '`()`', is a special type meaning *empty*. Representing an object with no value, it is a zero-width type, meaning a symbol of this type occupies no space. For examples of use, a unit is returned from a block which returns no value.

```
let a = {}; // a: ()
```

Note that a unit is neither an r– nor an l-value, so it cannot be used in expressions. That is, the below program errors before even trying to find a candidate for `+`.

```
() + 1; // error: expected rvalue, got ()
```

3.3 Pointer Types

The use of pointers is designed to be minimal in Edel, but they are available nonetheless. Simply put, a pointer stores the *location* (or *address*) of a variable, which can then be used to later reference and access that data. A pointer is indicated by a star '`*`' preceding the type; that is, type `*T` is telling the compiler *hey, see this location? Data of type* `T` *is stored there.*
Subsection 5.2 explains the use of the `&` and `*` unary operators to manipulate pointers.
Pointers do not fit into the type hierarchy, and cannot be implicitly cast. For safety, it is best not to cast pointer types unless one really knows and understands what is going on.

3.4 Casting

Casting refers to the changing of a type, and can be split into two categories: implicit, and explicit. Note that the destination type of either cast **cannot** be zero-sized.
Implicit casting is where a cast is done behind-the-scenes. Type a may be implicitly cast (also known as *coerced*) to another type b if and only if $a < b$, that is, a is a subtype of b. In other words, a can be coerced into b if b is sufficient to represent a without data loss. For example, `u8` may be coerced into `u64`, or `float` into `double`, but **not** `i32` into `i16`. Implicit casting is done automatically in the following:

```
let a: double = 3.14; // float -> double
```

On the other hand, explicit casting violates these rules, and hence the cast may be *explicit*. Note that not all datatypes may be cast between each-other using explicit casting, but the subtype relationships may be largely violated.

```
let a: u8 = 42; // error!
let a = 42 as u8; // i32 -> u8
```

4 Functions

5 Operators

Operators are special symbols which combine one or two expressions, known as unary and binary operators, respectively. The table below lists the built-in operators alongside their behaviour and associativity. Operators start from tightest to loosest precedence, with those of the same precedence grouped.

5.1 Casting

The concept of casting is explained in subsection 3.4. Note that the table above presents two casting operators: `(T) a` and `a as T`.
The former is a *primitive* cast, meaning it only supports casting to primitive, atomic types. This cast is direct and is ignorant of the type of `a` and whether the cast is meaningful.
The latter, using the `as` keyword, is a straightforward explicit cast. There are no restrictions on `T` and, as explained in subsection 3.4, this cast still must obey some rules.

```
let a = (int) 3.14; // ok
let b = (*int) 0; // error
let c = 0 as *int; // ok
```

Also note the difference in precedence: the `as` cast is slightly looser than other unary operators.

```
!1 as bool; // same as '(!1) as bool' NOT '!(1 as bool)'
```

5.2 Address-Of & Dereferencing

Both operations involve pointers, with the former generating pointers, and the latter 'consuming' them.
The address-of operator, `&`, calculates the memory address of a symbol. That is, if `a: T`, then `&a` is of type `*T`.
Dereferencing, then, is address-of's twin; it retrieved the value stored at a pointer. That is, if `p: *T`, then `*a` is of type `T`.

Table 2: Built-In Operators

Operator	Description	Associativity	Overloadable
`a.b`	Member access	$\longrightarrow$	No
`a()`	Function call	$\longrightarrow$	Yes

`(T) a`
`&a`
`*a`
`registerof a`
	Primitive cast
	Address-of
	Dereference
	Register lookup
		$\longleftarrow$	No

`-a`
`~a`
`!a`
	Negation
	Bitwise NOT
	Boolean NOT
		$\longleftarrow$	Yes

`a as T`	Cast	$\longleftarrow$	No

`a * b`
`a / b`
`a % b`
	Multiplication
	Division
	Modulo (remainder)
		$\longrightarrow$	Yes

`a + b` `a - b`	Addition Subtraction	$\longrightarrow$	Yes
`a << b` `a >> b`	Bitwise left shift Bitwise left shift	$\longrightarrow$	Yes

`a == b`
`a != b`
`a < b`
`a <= b`
`a > b`
`a >= b`
	Relational operators	$\longrightarrow$	Yes

`a & b`	Bitwise AND	$\longrightarrow$	Yes
`a ^ b`	Bitwise XOR	$\longrightarrow$	Yes
`a | b`	Bitwise OR	$\longrightarrow$	Yes
`a = b`	Assignment	$\longleftarrow$	No

```
let a: int = 5;
let p: *int = &a;
let b: int = *p; // then a == b
```

5.3 Register Lookup

This is a lower-level operator meant for exploration of the processor's registers.
Expecting a symbol argument, the **`registerof`** operator returns the index (offset) of where the given symbol is located. This allows one to gain insight into the register allocation algorithm.
For example, given the basic program below, one would expect the following.

```
let a = 5;
registerof a; // = 11, corresponding to $r1
```

If the symbol has not been loaded into registers (i.e., it has not been used at all/recently), the operator returns -1.

```
let a: int;
registerof a; // = -1
```