Language Documentation

Ruben Saunders

January 2025

Contents

1 Introduction

This document will contain information about the Edel language. As an educational language, 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 (except variables) 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:

- `--always-define-symbols` - always define symbols even if they are unused (e.g., all functions).
- `--ast` - prints the parsed program's AST.
- `-d` - enables debug mode. This will append each line with a trace back to the source code. Note that this is required by the visualiser component.
- `--[no-]function-placeholder` - changes behaviour of a function declaration; see subsection 5.3.
- `--[no-]indentation` - adds/removes indentation from generated assembly.
- `--[no-]lint` - enables/disables linting messages.
- `--lint-level` n - sets the linting error reporting level.
    0. Note.
    1. Warning.
    2. Error.

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");
}
```

1.4 Linting

As this compiler focuses on ease-of-use, it does not contain many linting options or messages. However, some messages such as empty blocks/namespaces have been included in hopes to alert programmers of unintentional happenings.

2 Names & Scope

A *name* in Edel refers to an identifier, which includes functions, variables, and namespaces. All names have a *scope*, which is the area in which that name exists/may be referenced. That is, outside its scope, a symbol does not exist.

The outermost scope is known as the *global* scope. Names defined here will be available anywhere in the current file.

The counterpart is the *local* scope, which refers to the latest scope. New scopes are created by using code blocks, for example, in while loops and functions.

2.1 Variables

Using the common analogy, a variable is like a labeled box which stores 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 variable's 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 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`. As they cannot be re-assigned, a constant definition **must** be initialised.

```
const pi = 3.14159;
```

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

2.2 Namespaces

Namespaces allow the programmer to group related names in an environment. Not only does this aid in organisation, but also helps to prevent naming collisions. Note that this does **not** create a new lexical scope.

A namespace is created using the `namespace` keyword, and may contain any structure which is permitted at the global level.

```
namespace maths {
    func cos(x: double) -> double;
}
```

To reference a namespace's member outside the namespace, it must be prefixed by the namespace's name.

```
cos(0); // error: name 'cos' does not exist
maths.cos(0); // = 1
```

Unlike variables, namespaces cannot be shadowed. Note that the `namespace` keyword only creates a namespace if it does not already exist. If it does, then the declarations will be added to the existing namespace. For example, the below snippet would result in the definitions of the two functions `maths.sin` and `maths.cos`.

```
namespace maths {
    func sin(x: double) -> double;
}
namespace maths {
    func cos(x: double) -> double;
}
```

2.2.1 Nested Namespaces

Namespaces may be nested explicitly as follows.

```
namespace std {
    namespace maths {
        func cos(x: double) -> double;
    }
}
maths.cos; // error: name 'maths' does not exist
std.maths.cos; // ok
```

However, namespace names may contain paths, that is, multiple identifiers joined by the dot '.' operator. This would be equivalent to explicitly nesting each identifier as a namespace. For example, one may re-write the above snippet as

```
namespace std.maths {
    func cos(x: double) -> double;
}
```

2.3 The Discard Symbol, '_'

The discard symbol, '_', is a special identifier meant for instances where the result/value is intended to be discarded. It cannot be used as a general identifier, with its special uses outlined below. For example, it cannot be used as a function or namespace name.

2.3.1 In Let Statements

This symbol may be used in a let statement. In this case, an assignment is required, and the variable **cannot** have a type annotation. While the right-hand side will be evaluated, no assignment will take place.

```
let _: int; // error: expected assignment
let _: int = 1 + 2; // error: cannot have a type annotation
let _ = 1 + 2;
```

2.3.2 As a Parameter Name

Another use is for function parameters. Unlike ordinary parameters, this symbol may be re-used for multiple parameters. Then, as this symbol cannot be referenced, this is equivalent to "throwing away" the parameter.

```
func middle(_: int, x: int, _: int) -> int {
    return x;
}
```

3 Basic Syntax

Inspired 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.

3.1 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.

3.1.1 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 4, 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
```

3.1.2 Values are Copied

An important thing to note is that, when values are passed around, such as function arguments and during assignment, they are **copied**. Therefore, care should be taken to avoid accidental copying when it is neither desired nor required.

For example, note the following where the original array is not mutated as the argument is copied.

```
func useless(arr: [int; 5]) {
    arr[0] = 1;
}

let a: [int; 5]; // a[0] = 0
useless(a);
// a[0] = 0 still
```

If this mutation behaviour was desired, pointers should be used.

```
func useful(arr: *[int; 5]) {
    (*arr)[0] = 1;
}

let a: [int; 5]; // a[0] = 0
useful(a);
// a[0] = 1
```

3.2 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. As previously mentioned, a block may be an expression or a statement, hence is permitted anywhere they would be expected.

Blocks in Edel introduce a new lexical scope (see section 2 for scope resolution).

3.3 Literals

A literal is an rvalue represented directly in the code, such as integers and Booleans. As an rvalue, each literal has a type (see section 4).

3.3.1 Booleans

A Boolean is a binary value, with each Boolean being `true` or `false`.

```
true;
false;
```

3.3.2 Numbers

Numbers can be divided into two categories: integers, or whole numbers, and floating-point numbers. Both consists of a sequence of digits, with the latter containing a decimal point.

```
12;
3.14159;
```

The default type of each category is `int` and `float`, respectively. If a specific type is required, this may appear as a suffix. For example,

```
12u8;
3.1415f64;
```

Note that the type of a literal is determined lazily. That is, the type is only determined when the literal is required. For example, the following integer is parsed as a `u8` rather than as an `int` before being cast to a `u8`.

```
12 as u8;
```

3.3.3 Null

The `null` keyword is used to denote an empty pointer/function value. `null` can be through of as the following.

```
let null: *u8 = 0;
```

(Note this is not well-formed code, as the cast would fail, and `null` is a keyword.)

As with numeric literals, `null`'s type is also determined in-context. `null` may be used as any pointer type, as well as a functional type.

```
let ptr = null as *i32;
let fn = null as (int) -> int;
```

3.3.4 Arrays

Array literals represent static array types, such as `[int; 5]`. An array literal is a comma-seperated list of expressions enclosed in square brackets. The type of an array literal, if not explicitly provided, is taken from the type of the first element.

```
let a = [1]; // ok, [int; 1]
let b = [1, 3.4]; // error: cannot convert float to int
let c: [float; 2] = [1, 3.4]; // ok
```

4 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.

4.1 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.

4.2 The Unit Type

Ths unit type, denoted 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: ()
```

4.2.1 Zero-Sized Types

It has been mentioned that `()` is a zero-sized type, meaning it takes up no physical space. These types exist purely for type-checking and compile-time enforcement of constraints but do not contribute to a program's memory footprint.

```
operator +(a: int, b: ()) {
    return ();
}
let ans = 1 + (); // = ()
```

In the above snippet, as `a` has type `()`, it does not occupy any space on the stack. However, the assignment expression is still executed, despite the result of `()`, to ensure any side-effects are captured.

4.3 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 *this is a location, and data of type* `T` *is stored there.*

Subsection 6.1 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 absolutely necessary.

4.3.1 Pointer Arithmetic

Pointer arithmetic refers to the addition or subtraction of a pointer and an integer. In such cases, the integer is "scaled" by the size of the data pointer to by the pointer. That is, let `ptr` be a pointer of type `*T`, then

```
ptr + 2
// turns into
ptr + 2 * sizeof(T)
```

This is natural when considering `ptr` as a buffer of type `T`'s; `+1` would refer to the next item, which would be `sizeof T` bytes away.

Note that this only works for `pointer + scalar` and not vice versa. Furthermore, the integer must be, or be a subtype of, `u64`.

4.3.2 Pointer Subscripting

The subscript operator '`[]`' can be used on pointer types, in which case the subscript acts as a shorthand for pointer arithmetic. As `[]` notation is often used with arrays, this extends the behaviour of pointer arithmetic in regard to referring to the "next" item in a buffer. The following expressions are identical in behaviour.

```
ptr[i]
// is equivalent to
*(ptr + i)
```

4.4 Function Types

Every function has a function type, with its type describing its argument and return types. A function t ype consists of a bracket-enclosed, comma-separated list of argument types, followed by an arrow and the return type. Unlike in function definitions, an argument type list and return type are necessary. For example,

```
let a: (int) -> int;
let b: (int) -> ();
let c: () -> ();
```

A symbol with a functional type can then be called as if it were a function. However, before calling, the symbol **must** be assigned to. Unlike overloading, when assigning to a functional type, the functional type must be an exact match.

```
func abs(a: int) -> int {
    return if a > 0 { a } else { -a };
}

let f: (int) -> int = abs;
let result = f(-25);
```

4.5 Array Types

An array is a collection of n instances of a single type located contiguously in memory. Array types are enclosed in square brackets, containing the type, a semi-colon, then the array's size.

```
let a: [int; 5];
```

In the above snippet, a region of $5 \times 4 = 20$ bytes will be reserved.

Items in an array may be accessed using the subscript operator, '`[]`'. The dereference operator, '`*`', may be used to get the first element in an array.

4.5.1 Nested Arrays

Arrays may be nested by placing another array type in the square brackets of an array type.

```
let a: [[int; 2]; 5];
```

Due to copy semantics, be aware of the following:

```
let b = a[1]; // b: [int; 2]
```

The array `a[1]` is copied into `b`, as opposed to `b` holding a reference to it.

4.6 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, a cast which violates the type hierarchy must 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; // ok
```

4.6.1 Casting Constraints

Implicit casting is constrained by subtyping as dictated in the subtype graph. Explicit casting allows the programmer to circumvent many of these rules, but it is not a completely unconstrained process.

The constraints are as follows: say we have the following cast:

```
let a: T1;
a as T2;
```

This cast is invalid if:

- `T2` is a function type and neither of the following are true:
  - `T1` is a pointer; or
  - `T1` is also a functional type, and `T1 == T2`.
- `T2` is a pointer, but `T1` is not.

Circumvention These constraints exist to prevent runtime errors and unexpected behaviour. However, if the programmer *really* wishes to perform an illegal explicit cast, this may be done by adding an exclamation mark after the `as` keyword.

```
let ptr = 12 as *i32; // error
let ptr = 12 as! *i32; // ok
```

This should be avoided and is often a sign of bad code practice.

4.7 Type "Hints"

To explain this concept, consider the following situation: referring to an overloaded function. For example,

```
func add(a: int, b: int) -> int {
    return a + b;
}

func add(a: float, b: float) -> float {
    return a + b;
}

add; // error: unable to resolve overloaded symbol 'add'
```

This makes sense, as the compiler has no idea which overload of `add` the programmer wants. This is where type hints come in. In certain scenarios where the desired type is known, this context can act as a prompt to the compiler on which overload the programmer desires. For example, in the following, the programmer is assigning to a variable of type `(int, int) -> int`, so the compiler deduces which overload they want.

```
let f: (int, int) -> int = add;
```

This also occurs during casting:

```
add as (int, int) -> int;
```

And when determining the arguments to a non-overloaded function.

```
func apply(x: int, y: int, f: (int, int) -> int) -> int {
    return f(x, y);
}

apply(1, 2, add);
```

Note, this only works for non-overloaded functions, as otherwise the "target" type for each argument is not known. This is also useful when obtaining a pointer to an overloaded function, as `&` expects a symbol.

```
let ptr = &(add as (int, int) -> int); // error
let ptr = &add as *(int, int) -> int; // ok
```

5 Functions

Functions are labelled blocks of code which may be called by the programmer. A function may take some input values, known as *arguments*, and may optionally return a value.

An example of a function which takes two integers and returns an integer:

```
func add(a: int, b: int) -> int {
    return a + b;
}
```

A function is defined using the `func` keyword, followed by a name. Parameters in the form `name: type` are listed, comma-separated, between brackets. Note that if no arguments are required, the brackets may be omitted altogether. The function's return type is listed after an arrow; if no return type is needed, this may also be omitted. For example,

```
func foo { // signature food() -> ()
    // ...
}

func bar(a: float) { // signature foo(float) -> ()
    // ...
}
```

5.1 Calling Functions

Functions are called by following the function name with brackets '`()`' (this is the operator name). Any arguments to be supplied to the function are comma-separated inside these brackets, known as an *argument list*. If the function is non-overloaded (i.e., only one function with this name exists), the number of arguments must match, and each argument's type must also match the respective parameter type. Otherwise, if the function is overloaded, a function with a matching signature is then found (see subsection 5.4 on function overloading).

```
func add(a: int, b: int) -> int {
    return a + b;
}
let result = add(1, 4); // result: int = 5
```

Note that `()` is an operator like any other. Therefore, if the subject is not a function type, an overload for `operator()` is searched for. The signature will be the subject type followed by each argument's type. For example,

```
5(3.14, true); // operator()(i32, f32, bool)
```

5.2 Returning a Value

The `return` keyword is used to exit/return from the current function. Optionally, the keyword may be followed by an expression, in which case that value will be returned. The type of the returned expression must match, or be a subtype of, the function's return type. Furthermore, if the function returns a non-unit type, a return statement must be present.

```
func foo {
    return 42; // error: cannot convert i32 to ()
}

func bar -> int {
} // error: missing return statement
```

Note that *all* code paths should return, even if you, the programmer, knows that a path is never taken. That is because the compiler cannot currently recognise, for example, always-falsy conditions.

```
func foo -> int {
    if false {
    } else {
        return 42;
    }
} // error: missing return statement
```

5.2.1 Unreachable Code

As `return` exits the function, any code after said statement is *unreachable*. This code will never be executed, and a warning will be reported.

```
func foo {
    return;
    a; // warning: unreachable code detected
}
```

The above code would usually result in an error, as symbol '`a`' is not defined. However, as the code is not reached, no error is raised.

5.3 Function Declaration

A function declaration is one which has no body. For example,

```
func add(a: int, b: int) -> int;
```

As functions in Edel do not require forward-declaration, what does this do? Depending on the compiler flag `--function-placeholder`, it does one of two things.

Enforce Existence The default, `--no-function-placeholder`, a function declaration is used to inform the programmer about the existence of a function. One common use is in auto-generated standard library files, where the function exists but is implemented internally. If the function does not exist, an error will be raised.

```
func imaginary_function(x: int) -> bool; // error: function was declared but
   does not exist
```

Generate Placeholder In this instance, if a function is declared but does not exist, the function will be created to return zero, '`()`', false, or null, depending on the return type. This is useful when a function is yet to be implemented.

For example, the following are identical.

```
func imaginary_function(x: int) -> bool;
// ==
func imaginary_function(x: int) -> bool {
    return false;
}
```

Note that as this generates a definition, it cannot be re-defined.

5.4 Function Overloading

A function in Edel is not only identified by its name, but also its *signature*. The signature of a function is a combination of its name and parameter types. In an Edel program, a function signature must be unique, or the result would be an error. This means that multiple functions with the same name can exist. This is known as function overloading, and a variant of a function with the same name but different parameters is called an *overload*.

```
func add(a: int, b: int) -> int {
    return a + b;
}
func add(a: float, b: float) -> float {
    return a + b;
}
```

In this case, a reference to `add` no longer just refers to one symbol. Instead, it becomes what is known as an *overload set*, and more information is required for a resolution.

```
add; // error: unable to resolve symbol 'add'
```

5.4.1 Overload Resolution

This is the process by which an overload is selected. Firstly, a list of candidates is generated; this would be all functions with a matching name. A candidate is considered a match if the number of arguments match and the type of each parameter matches. Due to subtyping rules, the latter check may lead to the generating of multiple 'matching' candidates. In such a case, the candidate with the highest number of identical types wins.

```
func add(a: int, b: int) -> int;
func add(a: long, b: long) -> long;

add(1, 2); // signature is add(int, int)
```

In the above snippet, both candidates match the callee. However, the former wins as `int == int` while `int < long`.

If there are still ties at this point, an error is raised as a candidate cannot be decided. In this case, it is best to explicitly cast one or more arguments to reach the desired candidate.

```
func add(a: int, b: long) -> long;
func add(a: long, b: int) -> long;

add(1, 2); // error: unable to resolve symbol 'add'
```

6 Operators

Operators are special symbols which combine one or two expressions, known as unary and binary operators, respectively. In Edel, operators are generic, and any string containing the following characters is parsed as an operator:

```
!#$%&*+./<=>?@^|-
```

As operators are generic (rather than a pre-defined set of symbols), certain information is determined by its use. Namely, an operator's *arity* is determined by its placement: unary if it is seen preceding a term, binary otherwise.

An operator's *associativity* determines the direction in which a chain of said operators are evaluated. As an example, let $\sim$ be a binary operator in the expression $a \sim b \sim c$.

- Left-associative (ltr, $\longrightarrow$): equivalent to $(a \sim b) \sim c$.
- Right-associative (rtl, $\longleftarrow$): equivalent to $a \sim (b \sim c)$.

Finally, operators have a *precedence*. This is how tightly an operator binds, and is useful to avoid having to explicitly include brackets everywhere. For example, $a + b * c \equiv a + (b * c)$ but **not** $a + b) * c$ as `*` binds tighter than `+`.

The table below displays a list of built-in operators alongside their associativity and precedence. The associativity and precedence of a generic (non built-in) operator is also shown.

Table 2: Operator Precedence and Associativity

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

`&a`
`*a`
`sizeof a`
	Address-of
	Dereference
	Get size
		$\longleftarrow$	No

`-a`
` a`
`!a`
⟨*op*⟩`a`
	Negation
	Bitwise NOT
	Boolean NOT
	Generic
		$\longleftarrow$	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` ⟨*op*⟩ `b`	*Generic*	$\longrightarrow$	Yes
`a && b`	Logical AND	$\longrightarrow$	No
`a || b`	Logical OR	$\longrightarrow$	No
`a = b`	Assignment	$\longleftarrow$	No

6.1 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`.

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

The dereference operator returns an lvalue, so it may be assigned to. In this case, the memory pointed to by the pointer will be written to. Continuing the example from above,

```
*p = 10; // a = 10
```

However, `b` will still be 5 as it was assigned to `*p` previously and is no longer tied to the same address.

6.2 Sizeof

The `sizeof` operator returns the size, in bytes (as a `u64`), of its argument. Since the argument to `sizeof` can be either a type or an expression, the returned value corresponds to the size of the provided type or, if given an expression, the size of its type.

```
sizeof(int); // 4
sizeof(42); // 4
```

(The brackets are not required as `sizeof` is an operator and are shown for clarity only.)

6.3 Logical Operators

Operators '`&&`' and '`||`' act as logical AND and OR, respectively. That is, the former is true if and only if both arguments are true, and the latter if any of its arguments are true.

Both operators have the notable property of being *lazy*. This means that the second argument is only evaluated if necessary. If the first argument is sufficient to calculate the operator's result, i.e., `true` for `||` and `false` for `&&`, theen the operator is *short-circuited* and the second argument is skipped. For example, in neither of the below expressions, function `f` is never called.

```
true || f(); // == true
false && f(); // == false
```

6.4 Operator Overloading

Much like functions, operators may be overridden, but this also allows new, custom operators to be defined. An operator overload is defined much like a function, except starting with the `operator` keyword, followed by the operator symbol. For example,

```
operator $(x: int) -> int {
    return x * 100;
}
```

defines a new unary overload for the operator `$` when applied to an integer. Similarly, one may define a binary overload.

```
operator $(a: int, b: int) -> int {
    return $a + $b;
}
```

Given these definitions, one could then do the following.

```
let a = $5; // = 5 * 100 = 500
let b = 1 $ 2; // = 1 * 100 + 2 * 100 = 300
```

It is perfectly permissible to overload built-in operators. Just note that these operators have a non-standard associativity and precedence.

```
operator +(a: int, b: float) -> float {
    return (float) a + b;
}
let ans = 1 + 3.14; // = 4.14
```

Like functions, an existing operator overload cannot be shadowed. For example, the following would fail.

```
operator +(a: int, b: int) -> int { // error: operator+(i32, i32) already exists
    // ...
}
```

7 Control Flow

Control flow refers to code which may alter the otherwise-linear flow of code. Without control flow, all programs would be deterministic and extremely limiting.

7.1 If Statements

An if statement is a basic conditional: if the condition, or *guard*, is true, it executes one section of code, otherwise executed another. The syntax of an if statement is as follows.

```
if condition {
    // truthy code
} else {
    // falsy code
}
```

With `condition` being a Boolean value. Note that the `else` branch is optional and may be omitted if not required.

7.1.1 If Expressions

Much like blocks, if statements may also be used as an expression by omitting the last semicolon in the branches. When used this way, both branches must return values of the *exact* same type. As a result, an if expression must always include an `else` branch.

```
let x = if a > 0 { 1 } else { -1 };
```

Here, both branches return an `int`, so the expression is valid.

However, if a branch always returns, its evaluated type is ignored since execution will never reach the rest of the expression.

```
let x = if a > 0 {
    1
} else {
    return false;
}
```

In this case, the `else` branch does not produce a value – it causes the function to return instead. Since the `else` branch never completes normally, its return type does not affect the type-checking of the if expression, and the code remains valid.

7.1.2 Multiple Branches

The effect of multiple conditional branches may be emulated by stacking if-else statements and removing the braces around all but the final `else` branch.

```
if C1 {
    // ...
} else if C2 {
    // ...
} else {
    // ...
}
```

Note that if both `C1` and `C2` are true, as the former is tested first, only its block will be executed before the statement is exited; `C2` and its block will never be evaluated.

7.2 Unconditional Loops

A loop is a section of code which is executed repeatedly. In unconditional loops, the only way to exit the loop is using the `break` keyword, which is covered in section 7.2.1.

An unconditional loop is introduced using the `loop` keyword, followed by a *loop body*, which may be a single line or a block.

```
loop {
    i = i + 1;
}
```

Note that a loop **cannot** be an expression, and hence must evaluate to '`()`'.

```
loop {
    1
} // error: a loop should evaluate to unit '()', got int
```

7.2.1 Break & Continue

These two keywords are used to control the behaviour of a loop. Note that these only affect the immediate outer loop, and cannot be used to, for example, break out of a nested loop.

Break The `break` keyword allows the programmer to break/exit out of the loop. Technically, control is transferred to the end of the loop body.

```
let i = 0;
loop {
    i = i + 1;
    print(i);
    if (i == 5) break;
}
// output = 1, 2, 3, 4, 5
```

Continue On the other hand, `continue` transfers control to the start of the loop, skipping the rest of the loop body.

```
let i = 0;
loop {
    i = i + 1;
    if (i == 3) continue;
    print(i);
}
// output = 1, 2, 4, 5, ...
```

7.3 Conditional Loops

A conditional loop is a loop which only iterates as long as a condition is met. This is commonly known as a while loop.

```
let i = 0;
while i < 5 {
    i = i + 1;
    print(i);
}
// output = 1, 2, 3, 4, 5
```

Observe that this is equivalent to a `loop` with a conditional `break` at the beginning.

8 Modules & Imports

9 Standard Library