

# The Intention Language Reference

Anna Bukowska, Marek Kaput

July 10, 2018

# Contents

<b>Preface</b>	<b>iv</b>
<b>I The Intention Language</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Notational Conventions . . . . .	2
1.2 Compile-time and Run-time . . . . .	4
1.3 Program Structure . . . . .	4
1.4 Values, Types, Terms and Results . . . . .	5
1.5 Namespaces . . . . .	5
<b>2 Lexical structure</b>	<b>7</b>
2.1 Input format . . . . .	7
2.2 Special Lexical Productions . . . . .	8
2.3 Identifiers, Keywords and Operators . . . . .	9
2.4 Comments and White space . . . . .	10
2.5 Literals . . . . .	10

2.5.1	Numeric Literals . . . . .	10
2.5.2	String Literals . . . . .	11
<b>3</b>	<b>Statements and expressions</b>	<b>15</b>
3.1	Assignment statement . . . . .	16
3.2	Variable or item value expressions . . . . .	16
3.3	Literal expressions . . . . .	16
3.4	Block expressions . . . . .	17
3.5	Operator expressions . . . . .	18
3.5.1	Arithmetic operators . . . . .	18
3.5.2	Term comparison operators . . . . .	20
3.5.3	Result operators . . . . .	20
3.5.4	Typing errors . . . . .	21
3.5.5	Division by zero . . . . .	21
3.5.6	Integer overflow . . . . .	21
3.6	Call expressions . . . . .	21
3.7	Loops . . . . .	22
3.8	Conditionals . . . . .	22
3.9	Return expressions . . . . .	23
<b>4</b>	<b>Items</b>	<b>24</b>
<b>5</b>	<b>Modules and Assemblies</b>	<b>25</b>

<i>CONTENTS</i>	iii
<b>II The Intentio Standard Library</b>	<b>26</b>
<b>6 Introduction</b>	<b>27</b>
<b>A Influences</b>	<b>28</b>
<b>Bibliography</b>	<b>29</b>
<b>Index</b>	<b>30</b>

# Preface

We live in times of rapid emerging of new, modern programming languages. Some of them, like Rust[9], Go[8] or Swift[2], have proved that programming language styles have not settled down and there is still room for new ideas, especially for merging existing paradigms. Fundamentally, one can observe a shift from imperative programming to functional programming.

Despite all these changes, not all ideas get a chance to shine. Some of them are becoming forgotten and treated as esoteric. One of these is goal-oriented evaluation, with Icon[12] being one of its most *iconic* implementers. Authors of this document believe that Icon exposes some very interesting ideas and they made an attempt to recreate them in a new programming language: *Intentio*, named after Latin for *intention*.

This document is the primary reference for the Intentio programming language. It consists of three parts:

- Chapters that *semi-formally*<sup>1</sup> describe each language construct and their use.
- Chapters that *semi-formally* describe runtime services and standard library that are core part of the language.
- Various appendix chapters.

This document does not serve as a beginner-friendly introduction to the language.

---

<sup>1</sup>This document tries to maintain reasonably formal description of all items, but there are no guarantees all cases are described. As a fallback, the *intentioc*[3] compiler can be used as secondary reference.

## Goals

The primary goals when designing Intentio was for language to satisfy following constraints:

1. It should feature core concepts of Icon language: goal-oriented evaluation and generators
2. It should support Unicode character set
3. It should be fast and easy to build prototype applications, the language should be *ergonomic*<sup>2</sup> from developer perspective **and** *IDE friendly*<sup>3</sup>
4. It should feature rich capabilities in processing textual data

Our main goals were **not**:

1. The language should be general purpose language
2. Compilation time should be short
3. Memory usage of Intentio programs should be low
4. It should be easy to integrate with other languages

## Acknowledges

The structure and parts of content of this document are inspired by two existing language specifications which we believe are good examples to follow: The Rust Reference[10], Haskell 2010 Language Report[6] and The Go Programming Language Specification[8].

---

<sup>2</sup>Source code of Intentio programs should be concise, pleasant to write and easy to reason about.

<sup>3</sup>Source code of Intentio programs should be easily processable by text editors and analysis tools.

# Part I

## The Intentio Language

# Chapter 1

## Introduction

Intentio is domain specific, imperative programming language oriented for processing textual data. Intentio provides goal-oriented execution, generators, strong dynamic typing with optional type annotations, and a rich set of primitive data types, including Unicode strings, lists, arrays, maps, sets, arbitrary and fixed precision integers, and floating-point numbers. Intentio tries to incorporate ideas of the Icon[12] programming language into modern programming patterns.

This part defines the syntax of Intentio language and informal abstract semantics for the meaning of such programs. We leave as implementation detail how Intentio programs are manipulated, interpreted, compiled, etc. This includes all steps from source code to running program, programming environments and error messages. This also means that this document do not describe the reference compiler for Intentio language - *intentioc*[3].

### 1.1 Notational Conventions

#### Grammar

Throughout this document a BNF-style notational syntax is used to describe lexical structure and grammar:

$$nonterminal \rightarrow \text{terminal} \mid alternative$$



Following conventions are used for presenting productions syntax:

<b>token</b>	terminal symbol
<i>rule</i>	nonterminal symbol (in italic font)
( <i>pat</i> )	grouping
[ <i>pat</i> ]	optional (0 or 1 times)
( <i>pat</i> ) <sup>+</sup>	repetition (1 to <i>n</i> times)
( <i>pat</i> ) <sup>*</sup>	optional repetition (0 to <i>n</i> times)
<i>pat</i> <sub>1</sub> <i>pat</i> <sub>2</sub>	concatenation
<i>pat</i> <sub>1</sub>   <i>pat</i> <sub>2</sub>	alternative
<i>pat</i> <sub>\pat'</sub>	difference (symbols generated by <i>pat</i> except those generated by <i>pat'</i> )

## Parametrized productions

Some productions in Intention's grammar (like raw string literals) cannot be expressed using context-free grammar with finite number of productions. In order to reduce need of falling back to natural language, a concept of *parametrized productions* is used throughout this document.

A production  $A(x_1, x_2, \dots, x_n) \rightarrow B$ , parametrized over arguments  $x_1, x_2, \dots, x_n$ , defines different production for each combination of its arguments.

## Unicode productions

A few productions in Intention's grammar permit Unicode[11] code points outside the ASCII range. These productions are defined in terms of character properties specified in the Unicode standard, rather than in terms of ASCII-range code points. Intention compilers are expected to make use of new versions of Unicode as they are made available.

## Source code listings

Examples of Intention program fragments are given in fixed-width font:

```
fun main() {  
    x = 4; y = 3;  
    println("sum = " + (x + y));  
}
```

In some situations, there are *placeholders* in program fragments representing arbitrary pieces of Intention code. Placeholders are written in italics. By convention *e* will mean expressions, *d* - item declarations, *t* - types, etc.:

```
if e1 { e2 } else { e3 }
```

## 1.2 Compile-time and Run-time

Intention's semantics obey a *phase distinction* between compile-time and run-time<sup>1</sup>. Semantic rules that have a static interpretation govern the success or failure of compilation, while semantic rules that have a dynamic interpretation govern the behaviour of the program at run-time.

## 1.3 Program Structure

An Intention program is structured syntactically and semantically into five abstract levels:

1. At the topmost of each Intention program or library is an *assembly*. In compiled environments assembly is an unit of compilation, while in interpreted environments assembly is a whole set of loaded modules.
2. At the topmost of each assembly is a set of *modules*. Modules provide a way to control namespaces and to re-use software in larger programs. A particular source code file of Intention program consists of one module. Module structure is flat, there is no concept of submodule.
3. The top level of each module is a set of *item declarations*. An item is a component of module, such as a function, type definition or constant variable.

---

<sup>1</sup>In interpreter environments, compile-time would consist of syntactic analysis and linting.

4. Items which contain the real, executable code are built of *statements* and *expressions*. Statements express sequential, side-effect-full actions to carry out. Expression denotes how to evaluate a *term* and evaluating expression returns a *result*.
5. At the bottom level is Intentio's *lexical structure*. It describes how to build tokens - the most basic blocks of program's source code from sequences of characters in source file.

## 1.4 Values, Types, Terms and Results

A *value* is a representation of some entity which can be manipulated by a program. A *type* is a tag that defines the interpretation of value representation. Values and types are not mixed in Intentio. Values by itself are untyped, value's type is required to perform any kind of operation on value.

A *(value, type)* pair is called a *term*. Terms represent data yielded from evaluating expressions. Intentio is *strongly typed* so implicit type conversions do not exist in the language, but it is not prohibited to include casts in expressions semantics (thus `5 + 4.0` runs successfully).

Evaluating expressions may either succeed or fail. A tagged union of successfully evaluated result or failure with information describing what failed is called a *result*. Terms and results are the basic blocks of representing information in Intentio.

Following Haskell-style code listing describes relationships between these concepts:

```
data Value = ...
data Type = ...

newtype Term = (Value, Type)

data Result = Succ Term
           | Fail Term
```

## 1.5 Namespaces

There are three distinct namespaces in Intentio:

**Item namespace** Consists item and variable names.

**Module namespace** Consists of module names and import renames.

**Type namespace** Consists of type names.

There are no constraints on names belonging to particular namespace, therefore it is possible for name `Int` to simultaneously denote an item/variable, module and type.

# Chapter 2

## Lexical structure

This chapter describes the lexical structure of Intentio. Most of the details may be skipped in a first reading of the reference.

In this chapter all white space is expressed explicitly in syntax descriptions, there is no implicit space between juxtaposed symbols. Terminal characters represent real characters in program source code.

### 2.1 Input format

Intentio program source is interpreted as a sequence of Unicode code points encoded in UTF-8, though most grammar rules are defined in terms of printable ASCII code points.

Token stream of Intentio program is defined as:

$$\begin{aligned} lexprogram &\rightarrow [whitespace] ( token [whitespace] )^* \\ token &\rightarrow id \mid qid \mid keyword \mid operator \mid literal \end{aligned}$$

Intentio is *case sensitive* language and each code point is distinct; for instance, upper and lower case letters are different characters.

The NUL character (U+0000) may be not allowed in whole program source text.

If an UTF-8-encoded byte order mark (U+FEFF) is the first Unicode code point in program source text, it may be ignored. Byte order mark may be not allowed anywhere else in program source text.

## 2.2 Special Lexical Productions

Following productions define Unicode character sets which are used to define non pure ASCII productions. These productions do not have any semantic meaning themselves.

- *xidstart* and *xidcont* are sets of characters that have properties *XID\_start* and *XID\_continue* as in Unicode Standard Annex #31[4], these productions define valid identifier characters
- *any* is any single Unicode character with all implementation-specific character restrictions applied
- *eol* matches either line feed character `\n` (U+000A, Unix-style newline marker) or carriage return and then line feed characters `\r\n` (U+000D U+000A, Windows-style newline marker)
- *whitespacechar* matches any character that has the *Pattern\_White\_Space* Unicode property, namely:
  - horizontal tab `\t` (U+0009)
  - line feed `\n` (U+000A)
  - vertical tab (U+000B)
  - form feed (U+000C)
  - carriage return `\r` (U+000D)
  - space (U+0020)
  - next line (U+0085)
  - left-to-right mark (U+200E)
  - right-to-left mark (U+200F)
  - line separator (U+2028)
  - paragraph separator (U+2029)

Additionally:

*noneol* → *any*<sub>\eol</sub>

*decdig* → 0 | 1 | ... | 9

*bindig* → 0 | 1

*octdig* → 0 | 1 | ... | 7

*hexdig* → 0 | 1 | ... | 9 | A | B | ... | F | a | b | ... | f

## 2.3 Identifiers, Keywords and Operators

*id* → ( *xidstart* (*xidcont* | ')\* )\keyword

*qid* → *id* : *id*

*keyword* → and | break | const | do | else | enum | export  
 | fail | fun | if | impl | import | in | is | module  
 | not | or | return | struct | trait | type | while  
 | yield | \_

*operator* → + | - | \* | /  
 | ( | ) | [ | ] | { | } | ;  
 | == | < | <= | > | >= | === | !==  
 | =

An *identifier* consists of a "letter" or underscore followed by zero or more letters, digits, underscores, and single quotes. Simple, unqualified identifiers (*id*) are always resolved within current module and scope. In order to be able to specify which module identifier belongs to, in most places identifier may be prefixed with module name and ":" character to form a *qualified identifier* (*qid*).

*Keywords* are identifier-like tokens which have special meaning in the grammar, all of them are excluded from the *id* rule. *Operators* are another special tokens, these ones are formed from symbol characters. *keyword* and *operator* productions have no use in Intentio grammar definition, instead particular tokens are used.

Implementations that offer lints or warnings for unused parameters/variables/items are encouraged to suppress such warnings for identifiers beginning with underscore. This allows programmers to use `_arg` for a parameter that they expect to be unused.

## 2.4 Comments and White space

$$\begin{aligned} \text{whitespace} &\rightarrow ( \text{whitespacechar} \mid \text{comment} )^+ \\ \text{comment} &\rightarrow \# (\text{noneol})^* \text{eol} \end{aligned}$$

A *white space* is a non-empty sequence of white space characters or *comments*. Comments in Intentio are only line-based, there is no concept of block comment.

## 2.5 Literals

$$\text{literal} \rightarrow \text{integer} \mid \text{float} \mid \text{string}$$

### 2.5.1 Numeric Literals

$$\begin{aligned} \text{decimal} &\rightarrow \text{decdig} ( \text{decdig} \mid \text{--} )^* \\ \text{binary} &\rightarrow ( \text{0b} \mid \text{0B} ) \text{bindig} ( \text{bindig} \mid \text{--} )^* \\ \text{octal} &\rightarrow ( \text{0o} \mid \text{0O} ) \text{octdig} ( \text{octdig} \mid \text{--} )^* \\ \text{hexadecimal} &\rightarrow ( \text{0x} \mid \text{0X} ) \text{hexdig} ( \text{hexdig} \mid \text{--} )^* \\ \\ \text{integer} &\rightarrow \text{binary} \mid \text{octal} \mid \text{decimal} \mid \text{hexadecimal} \\ \\ \text{exponent} &\rightarrow ( \text{e} \mid \text{E} ) [ \text{+} \mid \text{-} ] ( \text{--} )^* \text{decimal} \\ \\ \text{float} &\rightarrow \text{decimal} \text{.} \text{decimal} [ \text{exponent} ] \\ &\mid \text{decimal} \text{exponent} \end{aligned}$$



A *numeric literal* is either an *integer literal* or *floating-point literal*. Integer literals may be given in decimal (the default), binary (prefixed by `0b` or `0B`), octal (prefixed by `0o` or `0O`) or hexadecimal notation (prefixed by `0x` or `0X`). Floating-point literals are always decimal. A floating literal must contain digits both before and after the decimal point. A `"_"` character is allowed inside numeric literals for visual separation of digit groups, for instance `476_981__109_528_` is equal to `476981109528`. Negative numeric literals are described grammatically, not lexically.

## 2.5.2 String Literals

$$string \rightarrow [stringmod] ( string' \mid charstring \mid rawstring \mid regexstring )$$

Intentionio features very flexible string literals syntax. String literals can be written in four forms (regular, character, raw and regular expression) that make the literal compile to one of three value types (`String`, `Char` or `Regex`). String literals can be also prefixed with few modifiers that alter literal value before compiling it to value.

### Regular String Literals

$$string' \rightarrow " ( any_{\setminus(" \mid \backslash)} \mid escseq )^* "$$

A *string literal* is a sequence of Unicode characters, typed either directly or via escape sequence. String literals are compiled to `String` terms.

### Character Literals

$$charstring \rightarrow c" ( any_{\setminus(" \mid \backslash)} \mid escseq ) "$$

A *character literal* is a single Unicode character string, typed either directly or via escape sequence. Character literals are compiled to `Char` terms. The implementation is required to verify that character literal makes only one character.

## Raw String Literals

$$\begin{array}{ll}
 \text{rawstring} & \rightarrow \text{r } \text{rawstring}'(0) \\
 \text{rawstring}'(n) & \rightarrow \text{" } \text{rawstring}''(n) \text{" } \\
 & | \text{\# } \text{rawstring}'(n+1) \text{\# } \\
 \text{rawstring}''(n) & \rightarrow \text{An } (\text{any})^* \text{ that does not contain} \\
 & \text{" followed by \# repeated } n \text{ times.}
 \end{array}$$

A *raw string literal* does not process any escape sequences. It starts with letter **r**, followed by zero or more repetitions of hash symbol (**#**) and double quote (**"**). The raw string body can contain any sequence of Unicode characters and is terminated only by another double quote (**"**) followed by the same number of hashes (**#**) that preceded the opening quote.

### Example 2.5.1: Raw string literals

<code>"foo"</code>	<code>==</code>	<code>r"foo"</code>	<code># foo</code>
<code>"\"foo\""</code>	<code>==</code>	<code>r#"foo"#"</code>	<code># "foo"</code>
<code>"x #\"# y"</code>	<code>==</code>	<code>r##"x #"# y"##</code>	<code># x #"# y</code>

## Regular Expression Literals

$$\text{regexstring} \rightarrow \text{x ( string' | rawstring )}$$

A *regular expression literal* is a string literal that represents a regular expression and is compiled to **Regex** term. The exact syntactic and semantic details of regular expressions in Intentio are implementation dependent.

## Modifiers

$$\text{stringmod} \rightarrow \text{t} \mid \text{u}$$

A *string literal modifier* is a special flag that, when enabled, adds a step of processing of the literal value before compiling it to Intentio term. The order of modifiers is respected, they are processed from left-most modifier to right-most one.

Because modifiers alter literal contents before its compiling, they can fundamentally change their meaning, for instance the literal `tc" x "` should successfully compile and evaluate to single character `"x"`.

Intentionio provides following modifiers:

- `t` - *trim*: The trim modifier removes all white space characters from both sides of the string literal value.
- `u` - *unindent*: The unindent modifier gets the white-space-only prefix of the string literal value and then removes it from each line of the value.

## Escape Sequences

```

charescseq  →  \' | \" | \n | \r | \t | \\ | \0
asciiescseq →  \x hexdig hexdig
unicodeescseq → \u{ (hexdig | _ )+ }

escseq      →  charescseq | asciiescseq | unicodeescseq

```

Some *escape sequences* are available in non-raw string literals. An escape starts with a backslash character (`\`) and continues with one of the following forms:

- An *8-bit code point escape sequence* starts with letter `x` and is followed by exactly two hex digits with value up to `0x7f`. It denotes an ASCII character with value equal to provided hex value. Higher values are not permitted because it is ambiguous whether they mean Unicode code points or byte values, though the implementation should accept them on lexical level for better user experience.
- A *24-bit code point escape sequence* starts with letter `u` and is followed by up to six hex digits surrounded by braces `"{"` and `"}"`. It denotes the Unicode code point equal to the provided hex value. The implementation should accept more digits on lexical level for better user experience.
- The *character escape sequences* are convenience shortcuts for 8-bit code point escape sequences. Following table describes exact translations:

Escape	Character	Unicode
\'	'	U+0027
\"	"	U+0022
\n	\n	U+000A
\r	\r	U+000D
\t	\t	U+0009
\\	\	U+005C
\0	NUL	U+0000

# Chapter 3

## Statements and expressions

This chapter describes syntax and semantics of Intentionio *expressions*. Intentionio is an expression language. This means that all forms of result-producing or effect-causing evaluation fall into uniform syntax category of expressions. Usually each kind of expression can nest within each other kind of expression, and rules for evaluation of expressions involve specifying both the result produced by the expression and the order in which its sub-expressions are themselves evaluated.

A *statement* is either an assignment or expression. Statements serve mostly to contain and explicitly sequence expression evaluation, forming *blocks*. Evaluating *expression statement* means evaluating contained expression.

$stmt$	$\rightarrow$	<i>assign</i>	(assignment statement)
		<i>expr</i>	(expression statement)
$expr$	$\rightarrow$	<i>idexpr</i>	(variable or item value expression)
		<i>literalexpr</i>	(literal expression)
		<i>blockexpr</i>	(block expression)
		<i>unopexpr</i>	(unary operator expression)
		<i>binopexpr</i>	(binary operator expression)
		( <i>expr</i> )	(parenthesized expression)
		<i>callexpr</i>	(call expression)
		<i>whileexpr</i>	(while loop)
		<i>ifexpr</i>	(conditional expression)
		<i>returnexpr</i>	(return expression)

An *expression* is an syntactic construct that *evaluates* to a term. It may either *succeed* or *fail* resulting in a result. During the evaluation, expression may perform *side effects*, for example it can mutate some state or perform execution jump. The meaning of each kind of expression dictates several things:

- Whether or not to evaluate the sub-expressions when evaluating the expression
- The order in which to evaluate the sub-expressions
- How to combine the sub-expressions' results to obtain the result of the expression

### 3.1 Assignment statement

$assign \rightarrow \text{id} := expr$  (variable assignment)

An *assignment statement* mutates already declared variable, or declares new variable in current scope.

### 3.2 Variable or item value expressions

$idexpr \rightarrow \text{qid} \mid \text{id}$

An identifier or qualified identifier used in expression context denotes either local variable or declared item.

### 3.3 Literal expressions

$literalexpr \rightarrow \text{literal}$

A *literal expression* consists of one of the literal tokens. It directly describes a number or string.

Literals in Intentio are always immutable, it is not possible to change value of the literal in-place.

### 3.4 Block expressions

$blockexpr \rightarrow braceblock$

$braceblock \rightarrow \{ [blockbody] \}$

$parenblock \rightarrow ( blockbody )$

$blockbody \rightarrow ( stmt ; )^* expr [ ; ]$

A *block expression* is a sequence of statements, followed by an expression. Each block introduces a new namespace scope, this means that variables introduced within block are in scope for only the block itself.

Block evaluates statements sequentially while each statement evaluates successfully. On first failing statement, block stops evaluation and its result is equal to the result of failing statement. Otherwise, if all statements succeeded, block evaluates to result of the last expression. If block is empty, its result is a succeeded *unit* term.

## 3.5 Operator expressions

<i>unopexpr</i>	→	<i>negexpr</i>   <i>notexpr</i>
<i>binopexpr</i>	→	<i>arithexpr</i>   <i>cmpexpr</i>   <i>boolexpr</i>
<i>negexpr</i>	→	- <i>expr</i>
<i>notexpr</i>	→	not <i>expr</i>
<i>arithexpr</i>	→	<i>expr</i> ( +   - ) <i>expr</i>   <i>expr</i> ( *   / ) <i>expr</i>
<i>cmpexpr</i>	→	<i>expr</i> <i>cmpop</i> <i>expr</i>
<i>cmpop</i>	→	==   !=   <   >   <=   >=   ===   !==
<i>boolexpr</i>	→	<i>expr</i> or <i>expr</i>   <i>expr</i> and <i>expr</i>

Table 3.1: Operator precedence

Precedence	Operators	Associativity
6	-x, not x	right-to-left
5	*, /	left-to-right
4	+, -	left-to-right
3	==, !=, <, >, <=, >=, ===, !==	left-to-right
2	and	left-to-right
1	or	left-to-right

### 3.5.1 Arithmetic operators

Arithmetic operators apply to numeric terms (and, rarely, strings). Evaluating arithmetic operations always succeeds, behaviour in erroneous situations (such as division by zero) is described below. Following table lists all arithmetic operations, along with essential properties.



Table 3.2: Arithmetic operators

Operator	Meaning	Applies to types
<code>-x</code>	negation	integers, floats
<code>+</code>	sum	integers, floats, strings
<code>-</code>	difference	integers, floats
<code>*</code>	product	integers, floats
<code>/</code>	quotient	integers, floats

### Negation

For integer and floating-point operand terms, the `-x` operator is defined as follows. Term type is not changed during evaluation.

`-x == 0 - x`

### Integer and floating-point binary operations

For `+`, `-`, `*` and `/` operators, following typing rules hold:

```

Int + Int -> Int
Float + Float -> Float
Float + Int -> Float      # == Float + Float(Int)
Int + Float -> Float      # == Float(Int) + Float

```

### String concatenation

Strings can be concatenated using `+` operator. Following typing rules hold:

```

String + String -> String  # concatenate two strings
Char + String -> String    # prepend char to string
String + Char -> String    # append char to string

```

### 3.5.2 Term comparison operators

Comparison operators compare two operands and yield either success if comparison passed, or failure.

Table 3.3: Term comparison operators

Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	greater than or equal to
<code>===</code>	strict equal to
<code>!==</code>	strict not equal to

In *Intention*, arguments of comparison operators can be of any type. The following total ordering among types is defined:

`Number (Int or Float) < Char < String < Regex`

Integers are comparable with integers in usual way. Floating-point numbers are comparable with floating-point numbers according to IEEE-754[1] rules. For `===` and `!==` operators comparing integer to floating-point number (and vice-versa) always fails, otherwise integer is converted to float before comparison.

Characters are compared using their byte representation. Strings and regular expressions are compared character by character.

TODO(Ania): what term is returned?

### 3.5.3 Result operators

TODO(Ania): not, and, or

### 3.5.4 Typing errors

Not all typing rules are defined, and hence using of undefined combinations is prohibited. Evaluating such expressions results in run-time type error panic.

Example erroneous situations: `Char + Char`, `Regex + Regex`, `String + Regex`.

### 3.5.5 Division by zero

For integer operands, division by zero results in run-time panic. The result of a floating-point division by zero is not specified beyond the IEEE-754[1] standard; whether a run-time panic occurs is implementation-specific.

### 3.5.6 Integer overflow

Operators `+`, `-`, `*` and `/` may legally overflow, resulting value exists and is deterministically defined by the signed integer representation on destination architecture, the operation, and its operands. Evaluation is not failing nor no panic is triggered as a result of overflow. Intention compiler may not optimize code under the assumption that overflow does not occur.

## 3.6 Call expressions

$$\begin{aligned} \text{call expr} &\rightarrow \text{expr} \text{ ( } [\text{callargs}] \text{ )} \\ \text{callargs} &\rightarrow \text{expr} \text{ ( } \text{ , } \text{expr} \text{ )}^* \text{ [ , ]} \end{aligned}$$

A *call expression* is made of an expression followed by a parenthesized, comma separated expression list. It invokes a function, providing zero or more arguments, evaluating to result of function invocation (either succeeded return term, or failure). If evaluation of any of function arguments fails, then function is not called, and instead whole expression yields argument failure.

**Example 3.6.1:** Trivial call expression

```
y = add(3, 4)
```

**Example 3.6.2:** Calling arbitrary expressions

```
fun f(x) { e1 }
fun g() { f; }

y = g()(5) # works same as y = f(5)
```

## 3.7 Loops

$$whileexpr \rightarrow \text{while } ( \text{parenblock} \mid expr ) \text{ braceblock}$$

A *while loop* begins by evaluating loop condition block/expression. If evaluating loop condition succeeds, the loop body block executes. If the condition fails, then while loop completes and returns succeeded unit term. The result value of condition block is ignored.

TODO(Ania): Example

## 3.8 Conditionals

$$ifexpr \rightarrow \text{if } ( \text{parenblock} \mid expr ) \text{ braceblock} \\ [ \text{else } ( \text{braceblock} \mid ifexpr ) ]$$

A *conditional expression* is a conditional branch in program control. Conditional expression is made of a condition block/expression, followed by a consequent block, any number of alternative conditions and blocks, and an optional trailing final alternative block. If a condition expression evaluates successfully, the consequent block is executed and any subsequent **else if** or **else** block is skipped. If a condition expression evaluation fails, the consequent block is skipped and any subsequent else if condition is evaluated. If all condition blocks fails then **else** block is executed. Result values of conditions in condition block are ignored. Conditional expression evaluates to the same value as the executed block, or succeeded unit if no block is evaluated.

TODO(Ania): Example

## 3.9 Return expressions

$$return\ expr \rightarrow \text{return } [expr]$$

A *return expression* terminates execution of enclosing function, making it return the result of evaluating provided expression. In case no expression is given, a *unit* is returned.

**Example 3.9.1:** Return expression

```
fun abs(a) {  
  if a < 0 {  
    return -a;  
  }  
  return a;  
}
```

## Chapter 4

### Items

## Chapter 5

# Modules and Assemblies

Modules and items are entirely determined at compile-time, remain fixed during execution, and may reside in read-only memory. This limitation does not apply to assemblies, it is allowed to provide mechanisms to dynamically compile, load and unload assemblies at run-time.

## Part II

### The Intentio Standard Library



# Chapter 6

## Introduction

This part defines the Intentio Standard Library (shortly *stdlib*), its contents, semantics and the *prelude* which is automatically imported in each Intentio program. This library provides essentials for building proper Intentio programs, some of which should be used by the language implementation through compiler intrinsics. For developer convenience it also provides common utilities which ease application development and make a standard for code interoperability. The standard library must be distributed with each implementation of the Intentio language.

# Appendix A

## Influences

Intentio is not particularly original language, having the language Icon as the main source of inspiration, but also borrowing design element from wide range of other sources. Some of these are listed below:

- Icon[12]: goal-oriented execution, generators
- Rust[9]: syntax, string literals
- Erlang[5]: syntax, modules
- Python[7]: syntax

# Bibliography

- [1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.
- [2] Apple Inc. The Swift Programming Language. <https://swift.org/>.
- [3] Anna Bukowska and Marek Kaput. *intentio* - The reference Intentio compiler. <https://github.com/intentio-lang/intentio>.
- [4] Mark Davis. Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax. Technical Report Version 9.0.0.
- [5] Ericsson AB. Erlang programming language. <http://www.erlang.org/>.
- [6] Simon Marlow. Haskell 2010 language report. <https://www.haskell.org/definition/haskell2010.pdf>.
- [7] Python Software Foundation. Python language reference, version 3.6. <https://docs.python.org/3.6/reference/>.
- [8] The Go Authors. The Go Programming Language Specification. <https://golang.org/ref/spec>, licensed under CC BY 3.0.
- [9] The Rust Project Developers. The Rust Programming Language. <https://doc.rust-lang.org/book/>.
- [10] The Rust Project Developers. The Rust Reference. <https://doc.rust-lang.org/reference/>.
- [11] The Unicode Consortium. The Unicode Standard. Technical Report Version 9.0.0, Unicode Consortium.
- [12] University of Arizona. The Icon Programming Language. <https://www2.cs.arizona.edu/icon/>.

# Index

- comment, [10](#)
- compile-time, [4](#)
- escape sequence, [13](#)
- expression, [16](#)
  - block expression, [17](#)
  - call expression, [21](#)
  - conditional expression, [22](#)
  - identifier, [16](#)
  - literal expression, [17](#)
  - operator expression, [18](#)
  - return expression, [23](#)
  - while loop, [22](#)
- identifier, [9](#)
  - qualified identifier, [9](#)
- keyword, [9](#)
- literal
  - numeric literal, [11](#)
    - floating-point literal, [11](#)
    - integer literal, [11](#)
  - string literal, [11](#)
    - character literal, [11](#)
    - modifier, [12](#)
    - raw string literal, [12](#)
    - regular expression literal, [12](#)
- namespace
  - item namespace, [6](#)
  - module namespace, [6](#)
  - type namespace, [6](#)
- operator, [9](#)
- result, [5](#)
- run-time, [4](#)
- statement, [15](#)
  - assignment statement, [16](#)
  - expression statement, [15](#)
- term, [5](#)
- type, [5](#)
- value, [5](#)
- white space, [10](#)