The Intentio Language Reference

Anna Bukowska, Marek Kaput

August 23, 2018

Contents

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 *semi-formally*[1] describes each language construct and its use.

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

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

[1] 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.

2. It should support Unicode character set
3. It should be fast and easy to build prototype applications, the language should be *ergonomic*[2] from developer perspective **and** *IDE friendly*[3]
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].

[2]Source code of Intentio programs should be concise, pleasant to write and easy to reason about.

[3]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 set of primitive data types, including Unicode strings, fixed precision integers, and floating-point numbers. Intentio tries to incorporate ideas of the Icon[12] programming language into modern programming patterns.

This document 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 does not describe the Intentio standard library nor 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 → `terminal` | *alternative*

Following conventions are used for presenting productions syntax:

`token`	terminal symbol
rule	nonterminal symbol (in italic font)
(pat)	grouping
$[pat]$	optional (0 or 1 times)
$(pat)^+$	repetition (1 to n times)
$(pat)^*$	optional repetition (0 to n times)
$pat_1\ pat_2$	concatenation
$pat_1 \mid pat_2$	alternative
$pat_{\setminus pat'}$	difference (symbols generated by pat except those generated by pat')

Parametrized productions

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

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

Unicode productions

A few productions in Intentio'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. Intentio compilers are expected to make use of new versions of Unicode as they are made available.

Source code listings

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

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

1.2 Compile-time and Run-time

Intentio's semantics obey a *phase distinction* between compile-time and run-time[1]. Semantic rules that have a static interpretation govern the success or failure of compilation, while semantic rules that have a dynamic interpretation govern the behavior of the program at run-time.

1.3 Program Structure

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

1. At the topmost of each Intentio program or library is an *assembly*. In compiled environments assembly is a 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 Intentio program consists of one module. Module structure is flat, there is no concept of the submodule.

3. The top level of each module is a set of *item declarations*. An item is a component of the module, such as a function, type definition or constant variable.

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*.

[1] In interpreter environments, compile-time would consist of syntactic analysis and linting.

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 the 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 the particular namespace, therefore it is possible for name `Int` to simultaneously denote an item/variable, module and type.

1.6 Intentio source file

file → [`UTF8BOM`]
module

`UTF8BOM` → `\u{EFBBBF}` (UTF-8 Byte Order Mark)

An Intentio source file describes a single module, the name of which is defined by source file name. Source files have extension `.ieo`. Filename `example.int` defines module named `example`.

Intentio source files must always be encoded in UTF8. The optional UTF8 byte order mark (lexical `UTF8BOM` production) can only occur at the beginning of the file and must be ignored by the parser.

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:

$$
\begin{array}{lcl}
\textit{lexprogram} & \rightarrow & [\textit{whitespace}]\ (\ \textit{token}\ [\textit{whitespace}]\ )^{*} \\
\textit{token} & \rightarrow & \textit{id}\ |\ \textit{qid}\ |\ \textit{keyword}\ |\ \textit{operator}\ |\ \textit{literal}
\end{array}
$$

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*$_{\setminus eol}$

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* | `'`)*)$_{\setminus keyword}$
qid → *id* `:` *id*

keyword → `and` | `break` | `const` | `do` | `else` | `enum` | `export`
| `fail` | `fun` | `if` | `impl` | `import` | `in` | `is` | `module`
| `none` | `not` | `or` | `return` | `struct` | `succ` | `trait`
| `type` | `while` | `xor` | `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 the 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 hints or warnings for unused parameters/variables/items are encouraged to suppress such warnings for identifiers beginning with an underscore. This allows programmers to use `_arg` for a parameter that they expect to be unused.

2.4 Comments and White space

whitespace → (*whitespacechar* | *comment*)$^+$
comment → `#` (*noneol*)* *eol*

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

literal → *integer* | *float* | *string* | `none`

2.5.1 Numeric Literals

decimal → *decdig* (*decdig* | `_`)*
binary → (`0b` | `0B`) *bindig* (*bindig* | `_`)*
octal → (`0o` | `0O`) *octdig* (*octdig* | `_`)*
hexadecimal → (`0x` | `0X`) *hexdig* (*hexdig* | `_`)*

integer → *binary* | *octal* | *decimal* | *hexadecimal*

exponent → (`e` | `E`) [`+` | `-`] (`_`)* *decimal*

float → *decimal* `.` *decimal* [*exponent*]
| *decimal exponent*

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 (prefixed by `0x` or `0X`) notation. 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`. Regardless of the input method, only the value is stored. This means that after number is printed, no underlines will be displayed on the screen. Negative numeric literals are described grammatically, not lexically.

2.5.2 String Literals

string → [*stringmod*] (*string′* | *charstring* | *rawstring* | *regexstring*)

Intentio 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′ → `"` (*any*$_{\setminus(\,\texttt{"}\ \mid\ \setminus)}$ | *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 → `c"` (*any*$_{\setminus(\,\texttt{"}\ \mid\ \setminus)}$ | *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

rawstring	→	`r` *rawstring′*(0)
rawstring′(n)	→	`"` *rawstring″*(n) `"`
	|	`#` *rawstring′*($n+1$) `#`
rawstring″(n)	→	An (*any*)* that does not contain `"` followed by `#` repeated n times.

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 proceeded the opening quote.

Example 2.5.1: Raw string literals

```
"foo"        == r"foo"              # foo
"\"foo\""    == r#""foo""#          # "foo"
"x #\"# y"   == r##"x #"# y"##      # x #"# y
```

Regular Expression Literals

regexstring → `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

stringmod → `t` | `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"`, while `ct" x "` would report an error because there is not possible to make character from a multi-letter string.

Intentio 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

2.5.3 None Literal

The keyword none evaluates to successful none term, which is a special value none of type none. It is used to make boolean result as succ none or fail none.

Chapter 3

Modules and Assemblies

A *module* is a collection of zero or more items, declared in an environment created by a set of *imports* (items brought into scope from other modules). Module *exports* some of its items, making them available to other modules. The module corresponds to single *compilation unit*.

$$module \rightarrow exportdecl\ (itemdecl)^*$$

An *assembly* is a collection of one or more modules compiled together into single *compilation target*.

An Intentio *program* is an executable assembly, where one module is designated to be a *main module*. The main module must export a 0-arity function called `main`. This function is an *entry-point* or *main function* of the program. An the entry point is not required to be exported by main module. When the program is executed, this function is called and, and returned term is discarded. Depending on the running environment, the result state of the function call may be interpreted as program success or failure.

An Intentio *library* is an assembly containing reusable modules for use in other programs.

Modules are entirely determined at compile-time, remain fixed during execution, and may reside in read-only memory. This limitation does not apply to assemblies

(including single module assemblies). It is possible to provide mechanisms to dynamically compile, link, load and unload assemblies at run-time.

Example 3.0.1: Simple Module

```
# File: simple_module.ieo

export (compute, hello)

import math

fun compute(x) {
  x * math:sin(math:pi / x);
}

fun hello() {
  println("Hello World!");
}
```

3.1 Export declaration

exportdecl → `export` `(` [*exportspec*] `)`
exportspec → `id` (`,` `id`)* [`,`]

An *export declaration* is a list of names of all items that are exported by this module. A module may *re-export* items imported from other modules.

Example 3.1.1: Re-exporting

```
# File: myprelude.ieo

export (open, writeln, sin, pi)

import io:open
import io:writeln
import math:sin
import math:pi
```

3.2 Prelude

Each Intentio module implicitly imports all items from a *prelude* module (each module has an implicit `import prelude:*` at the very top). The name of the prelude module and its contents are implementation specific.

3.3 Module resolution

The exact behavior of module resolution algorithm is implementation-dependent, but typically a list of directories is scanned for a file with the same name as the imported module, plus the `.ieo` extension. The first directory on the list should always be a directory of importing module source file. The list may be specified by environment variable or command line option passed to the compiler.

Chapter 4

Items

This chapter describes the syntax and informal semantics of Intentio items. An *item* is a component of a module. Items are uniquely named within the module and are organized in the flat structure.

itemdecl	→	*fundecl*	(function)
	|	*importdecl*	(import declaration)

4.1 Functions

fundecl	→	`fun` `id` `(` [*funparams*] `)` *braceblock*
funparams	→	`expr` (`,` `expr` `expr`)* [`,`]

A *function* is a named block, along with optional set of parameters. Functions are declared with the keyword `fun`. If keyword *return* is not used, the function returns the result of evaluating the last expression from the contained block. Functions may declare a set of input variables as parameters, through which the caller passes arguments into the function. Input variables behave the same as normal variables defined within a function block.

A keyword `expr` before argument means that it would be evaluated. If it fails, the function will not be executed and return fail *none* term.

The number of function parameters n is called the *arity* of the function.

Example 4.1.1: Simple function

```
fun the_hardest_calculation(x) {
  x * 2
}
```

4.2 Import declarations

importdecl → `import` *importspec*

importspec	→	`id`	(qualified import)
	|	`id` `as` `id`	(renamed qualified import)
	|	`id` `:` `id`	(item import)
	|	`id` `:` `id` `as` `id`	(renamed item import)
	|	`id` `:` `*`	(import-all)

An *import declaration* creates local name binding in the current scope, synonymous with the specified module. Imports can import exported items directly into current scope or they can be *qualified* where the whole module is imported and exported items can be accessed using *qualified identifier* syntax.

Import declarations support five forms:

Qualified import
: Binding target module to its name:
`import a`

Renamed qualified import
: Binding target module to new local name:
`import a as b`

Item import
: Binding exported item of target module to its name:
`import a:x`

Renamed item import
: Binding exported item of target module to new local name:
`import a:x as y`

Import-all
: Binding all exported items of target module to their names:
`import a:*`

An import declaration declares a dependency relation between the importing and imported module. Modules may be mutually recursive, though module must not import itself, directly or indirectly.

Example 4.2.1: Import declarations

```
import io
import math:sin
import math as m

fun main() {
  f = io:open("result.txt", "w");
  io:writeln(f, sin(m:pi));
}
```

Chapter 5

Statements and expressions

This chapter describes the syntax and semantics of Intentio *expressions*. Intentio 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 encapsulate and explicitly sequence expression evaluation, forming *blocks*. Evaluating *expression statement* means evaluating encapsulated expression.

stmt	→	*assign*	(assignment statement)
	|	*expr*	(expression statement)
expr	→	*idexpr*	(variable or item value expression)
	|	*literalexpr*	(literal expression)
	|	*braceblockexpr*	(block expression)
	|	*unopexpr*	(unary operator expression)
	|	*binopexpr*	(binary operator expression)
	|	`(` *expr* `)`	(parenthesized expression)
	|	*callexpr*	(call expression)
	|	*whileexpr*	(while loop)
	|	*ifexpr*	(conditional expression)
	|	*returnexpr*	(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

5.1 Assignment statement

assign → `id` `=` *expr* (variable assignment)

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

5.2 Variable or item value expressions

idexpr → `qid` | `id`

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

5.3 Literal expressions

literalexpr → `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 the value of the literal in-place.

5.4 Block expressions

blockexpr → *braceblock*

braceblock → `{` [*blockbody*] `}`
parenblock → `(` [*blockbody*] `)`

blockbody → (*stmt* `;`)* *expr* [`;`]

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

A parenthesized block evaluates statements sequentially while each statement evaluates successfully. On the first failing statement, the block stops evaluation and its result is equal to the result of the failing statement. Otherwise, if all statements succeeded, the block evaluates to result of the last expression. If the block is empty, its result is a succeeded *none* term.

A braced block also evaluates statements sequentially, but it does not stop to evaluate statements even if some of them fails. Its result is always equal to the result of the last expression.

5.5 Operator expressions

unopexpr → *negexpr* | *notexpr*
binopexpr → *arithexpr* | *cmpexpr* | *boolexpr*

negexpr → `-` *expr*
notexpr → `not` *expr*

arithexpr → *expr* (`+` | `-`) *expr*
| *expr* (`*` | `/`) *expr*

cmpexpr → *expr cmpop expr*
cmpop → `==` | `!=` | `<` | `>` | `<=` | `>=` | `===` | `!==`

boolexpr → *expr* `or` *expr*
| *expr* `and` *expr*

Table 5.1: 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

5.5.1 Arithmetic operators

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

Table 5.2: Arithmetic operators

Operator	Meaning	Applies to types
`-x`	negation	integers, floats
`+`	sum	integers, floats, strings
`-`	difference	integers, floats
`*`	product	integers, floats
`/`	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:

```
Char + Char -> String       # concatenate two Chars
   making a String
String + String -> String   # concatenate two Strings
Char + String -> String     # prepend Char to String
String + Char -> String     # append Char to String
```

5.5.2 Term comparison operators

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

Table 5.3: Term comparison operators

Operator	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`===`	strict equal to
`!==`	strict not equal to

In Intentio, 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 the 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. For `===` and `!==` operators comparisons between characters, strings and regular expressions always fail, otherwise characters and regular expressions are always converted to strings before `==` and `!=` comparison.

No matter what comparison was made, its result consists of term copied from the second argument.

5.5.3 Result operators

Result operators are used to making logical expressions, such as conjunction, alternative, xor and negation. Their actions are analogous to those from which they took their name in classical mathematics.

The following tables present the exact results for each set of function arguments. The descriptions in the left column correspond to the first argument, while those in the top column - to the second. The results term is copied from the second argument.

Table 5.4: *and*

	Succ Term2	Fail Term2
Succ Term1	Succ Term2	Fail Term2
Fail Term1	Fail Term2	Fail Term2

Table 5.5: *or*

	Succ Term2	Fail Term2
Succ Term1	Succ Term2	Succ Term2
Fail Term1	Succ Term2	Fail Term2

Table 5.6: *xor*

	Succ Term2	Fail Term2
Succ Term1	Fail Term2	Succ Term2
Fail Term1	Succ Term2	Fail Term2

Table 5.7: *not*

Succ Term	Succ Term
Fail Term	Fail Term

5.5.4 Typing errors

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

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

5.5.5 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.

5.5.6 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. Intentio compiler may not optimize code under the assumption that overflow does not occur.

5.6 Call expressions

callexpr → *expr* `(` [*callargs*] `)`
callargs → *expr* (`,` *expr*)* [`,`]

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 the evaluation of any of function arguments fails, then the function is not called, and instead of whole expression yields argument failure.

Example 5.6.1: Trivial call expression

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

Example 5.6.2: Calling arbitrary expressions

```
fun f(x) { x * 2; }
fun g() { f; }

y = g()(5); # works same as y = f(5)
y == 10;
```

5.7 Loops

whileexpr → `while` (*parenblock* | *expr*) *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 *none* term. The result value of condition block is ignored.

Example 5.7.1: While expression with parentheses

```
let i = 0;
while (i < 8; i = i + 1) {
  println(i);
}
```

5.8 Conditionals

ifexpr → `if` (*parenblock* | *expr*) *braceblock*
[`else` (*braceblock* | *ifexpr*)]

A *conditional expression* is a conditional branch in program control. A 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 the condition is evaluated. If all condition blocks fail then `else` block is executed. Result values of conditions in condition block are ignored. A conditional expression evaluates to the same value as the executed block, or succeeded *none* if no block is evaluated.

Example 5.8.1: Trivial if expression

```
if a < b { c = a }
```

The variables defined in the condition could be used in the if-braceblock and the else-braceblock as it was the same scope.

Example 5.8.2: Trivial if expression with else (version1)

```
if (let a = 3; a < b) { c = a }
else { c = b }
```

which is equivalet to

Example 5.8.3: Trivial if expression with else (version2)

```
if (let a = 3; c = b; a < b) { c = a }
```

5.9 Return expressions

returnexpr → `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 *none result* is returned.

Example 5.9.1: Return expression

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

Part II

Appendices

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. *intentioc* - 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