The Intentio Language Reference

Anna Bukowska, Marek Kaput

June 3, 2018

Contents

Preface

We live in times of rapid emerging of new, modern programming languages. Some of them, like Rust[8], Go[7] or Swift[1], 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[11] 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*[1] 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.

[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*[2] 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*[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[9] and Haskell 2010 Language Report[5].

[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 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[11] 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*[2].

1.1 Notational Conventions

Grammar

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

$$\textit{nonterminal} \;\rightarrow\; \texttt{terminal} \mid \textit{alternative}$$

Following conventions are used for presenting productions syntax:

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

Parameterized productions

Some productions in Intentio'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 *parameterized productions* is used throughout this document.

A production $A(x_1, x_2, ..., x_n) \to B$, parameterized 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[10] 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, y := 4, 3;
  println(f"sum = ${x + y}");
}
```

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

`if` e_1 `{` e_2 `} else {` e_3 `}`

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 behaviour 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 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 Intentio 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.

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

4. Items which contain the real, executable code are built of *expressions*. 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:

```haskell
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:

$$
\begin{array}{lcl}
\mathit{lexprogram} & \rightarrow & \{\mathit{whitespace}\}\ \{\ \mathit{token}\ \mathit{whitespace}\ \} \\
\mathit{token} & \rightarrow & \mathit{id} \mid \mathit{keyword} \mid \mathit{operator} \mid \mathit{literal}
\end{array}
$$

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[3], 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:

$$\begin{array}{lcl}
noneol & \rightarrow & any_{\setminus(eol)} \\
 & & \\
decdig & \rightarrow & 0 \mid 1 \mid \ldots \mid 9 \\
bindig & \rightarrow & 0 \mid 1 \\
octdig & \rightarrow & 0 \mid 1 \mid \ldots \mid 7 \\
hexdig & \rightarrow & 0 \mid 1 \mid \ldots \mid 9 \mid A \mid B \mid \ldots \mid F \mid a \mid b \mid \ldots \mid f
\end{array}$$

2.3 Identifiers, Keywords and Operators

$$\begin{array}{lcl}
id & \rightarrow & (\ xidstart\ \{xidcont \mid \text{'}\}\ )_{\setminus(keyword)} \\
qid & \rightarrow & id\ :\ id \\
 & & \\
keyword & \rightarrow & \texttt{abstract} \mid \texttt{and} \mid \texttt{break} \mid \texttt{case} \mid \texttt{const} \mid \texttt{continue} \\
 & \mid & \texttt{do} \mid \texttt{else} \mid \texttt{enum} \mid \texttt{export} \mid \texttt{fail} \mid \texttt{fun} \mid \texttt{if} \mid \texttt{impl} \\
 & \mid & \texttt{import} \mid \texttt{in} \mid \texttt{is} \mid \texttt{loop} \mid \texttt{module} \mid \texttt{not} \mid \texttt{or} \mid \texttt{return} \\
 & \mid & \texttt{struct} \mid \texttt{type} \mid \texttt{where} \mid \texttt{while} \mid \texttt{yield} \mid \texttt{\_} \\
 & & \\
operator & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
 & \mid & \texttt{(} \mid \texttt{)} \mid \texttt{[} \mid \texttt{]} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{:} \mid \texttt{;} \\
 & \mid & \texttt{==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & \mid & \texttt{:=} \mid \texttt{<-} \mid \texttt{\$} \mid \texttt{\%}
\end{array}$$

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

$$\begin{array}{lcl}
\textit{whitespace} & \rightarrow & \langle\ \textit{whitespacechar} \mid \textit{comment}\ \rangle \\
\textit{comment} & \rightarrow & \#\ \{\textit{noneol}\}\ \textit{eol}
\end{array}$$

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

$$\begin{array}{lcl}
\textit{literal} & \rightarrow & \textit{integer} \mid \textit{float} \mid \textit{string}
\end{array}$$

2.5.1 Numeric Literals

$$\begin{array}{lcl}
\textit{decimal} & \rightarrow & \textit{decdig}\ \{\ \textit{decdig} \mid \texttt{\_}\ \} \\
\textit{binary} & \rightarrow & (\texttt{0b} \mid \texttt{0B})\ \textit{bindig}\ \{\ \textit{bindig} \mid \texttt{\_}\ \} \\
\textit{octal} & \rightarrow & (\texttt{0o} \mid \texttt{0O})\ \textit{octdig}\ \{\ \textit{octdig} \mid \texttt{\_}\ \} \\
\textit{hexadecimal} & \rightarrow & (\texttt{0x} \mid \texttt{0X})\ \textit{hexdig}\ \{\ \textit{hexdig} \mid \texttt{\_}\ \} \\
 & & \\
\textit{integer} & \rightarrow & \textit{binary} \mid \textit{octal} \mid \textit{decimal} \mid \textit{hexadecimal}
\end{array}$$

$$\begin{array}{lcl}
\textit{exponent} & \rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \{\texttt{\_}\}\ \textit{decimal} \\
 & & \\
\textit{float} & \rightarrow & \textit{decimal}\ \texttt{.}\ \textit{decimal}\ [\textit{exponent}] \\
 & \mid & \textit{decimal exponent}
\end{array}$$

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

$$\mathit{string} \quad \rightarrow \quad [\mathit{stringmod}]\ (\ \mathit{string'}\ \mid\ \mathit{charstring}\ \mid\ \mathit{rawstring}\ \mid\ \mathit{regexstring}\ )$$

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

$$\mathit{string'} \quad \rightarrow \quad \texttt{"}\ \{\ \mathit{any}_{\backslash(\ \texttt{"}\ \mid\ \backslash\ )}\ \mid\ \mathit{escseq}\ \}\ \texttt{"}$$

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

$$\mathit{charstring} \quad \rightarrow \quad \texttt{c"}\ (\ \mathit{any}_{\backslash(\ \texttt{"}\ \mid\ \backslash\ )}\ \mid\ \mathit{escseq}\ )\ \texttt{"}$$

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

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

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 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

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

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

$$
\textit{stringmod} \rightarrow \texttt{t}\ |\ \texttt{u}
$$

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"`.

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

Chapter 3

Expressions

This chapter describes 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.

Intentio does not have a concept of *statement* known from other programming languages.

$$
\begin{array}{lcll}
expr & \to & qidexpr & \text{(variable value or item)} \\
 & | & literalexpr & \text{(literal expression)} \\
 & | & blockexpr & \text{(block expression)} \\
 & | & unopexpr & \text{(unary operator expression)} \\
 & | & binopexpr & \text{(binary operator expression)} \\
 & | & \texttt{(}\ expr\ \texttt{)} & \text{(parenthesized expression)} \\
 & | & callexpr & \text{(call expression)} \\
 & | & loopexpr & \text{(loops)} \\
 & | & ifexpr & \text{(conditionals)} \\
 & | & returnexpr & \text{(return expression)}
\end{array}
$$

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 Variables and Items

3.2 Literal expressions

3.3 Block expressions

blockexpr	→	*braceblock*	(block expression)
block	→	*braceblock*	(block used as part of other expressions)
braceblock	→	`{` {*expr* `;`} `}`	

A *block expression* is a possibly empty sequence of expressions. Each block introduces a new namespace scope, this means that variables introduced within block are in scope for only the block itself.

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

3.4 Operator expressions

3.5 Call expressions

3.6 Loops

3.7 Conditionals

3.8 Return expressions

$$\textit{returnexpr} \quad \rightarrow \quad \texttt{return}\ [\textit{expr}]$$

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 3.8.1: 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 implementation through compiler intrinsics. For developer convenience it also provides common utilities which ease application development and make a standard for code interoperation. 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[11]: goal-oriented execution, generators
- Rust[8]: syntax, string literals
- Erlang[4]: syntax, modules
- Python[6]: syntax

Bibliography

[1] Apple Inc. The Swift Programming Language. https://swift.org/.

[2] A. Bukowska and M. Kaput. *intentioc* - The reference Intentio compiler. https://github.com/intentio-lang/intentio.

[3] M. Davis. Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax. Technical Report Version 9.0.0.

[4] Ericsson AB. Erlang programming language. http://www.erlang.org/.

[5] S. Marlow. Haskell 2010 language report. https://www.haskell.org/definition/haskell2010.pdf.

[6] Python Software Foundation. Python language reference, version 3.6. https://docs.python.org/3.6/reference/.

[7] The Go Authors. The Go Programming Language Specification. https://golang.org/ref/spec.

[8] The Rust Project Developers. The Rust Programming Language. https://doc.rust-lang.org/book/.

[9] The Rust Project Developers. The Rust Reference. https://doc.rust-lang.org/reference/.

[10] The Unicode Consortium. The Unicode Standard. Technical Report Version 9.0.0, Unicode Consortium.

[11] University of Arizona. The Icon Programming Language. https://www2.cs.arizona.edu/icon/.

Index