The Pragmatic Language

Language Primer

This document is designed to help programmers learn the basics of Bill.

Copyright (c) 2022 J. Smith. Permission is granted to copy, distribute and/or modify this document, provided this notice is kept.

Contents

**1 Introduction** **1**
1.1 Reserved Words . . . 2

**2 Types** **3**
2.1 Integers . . . 3
2.1.1 Unsigned . . . 4
2.2 Floats . . . 4
2.3 Strings . . . 4
2.4 Boolean . . . 4
2.5 Null . . . 4
2.6 Constants . . . 4
2.7 Others . . . 5

**3 Expressions** **7**
3.1 Math . . . 7
3.1.1 Operators . . . 8
3.2 Logic . . . 8
3.2.1 Operators . . . 9
3.3 Concatenation . . . 9
3.3.1 Operators . . . 9

**4 Containers** **11**
4.1 Arrays . . . 11
4.2 Vectors . . . 11
4.3 Sets . . . 12
4.3.1 Operators . . . 12
4.4 Tuples . . . 13
4.5 Dictionaries . . . 13
4.6 Others . . . 13

**5 Flow Control** **15**
5.1 Conditions . . . 15
5.1.1 Conditionals . . . 15
5.1.2 Loops . . . 16
5.2 Traversal . . . 17

**6 Functions** **19**
6.1 Calling a Function . . . 19
6.2 Declaring a Function . . . 19
6.3 Return . . . 20
6.4 Used in Expressions . . . 20
6.5 Built-in . . . 20

6.6 Nested . . . . . 21

**7 Exceptions** **23**

7.1 Throw . . . . . 23

7.2 Try . . . . . 23

7.3 Catch . . . . . 24

**8 Style** **25**

8.1 Comments . . . . . 25

8.2 Statements . . . . . 25

8.3 Blocks . . . . . 25

**Index** **27**

# Chapter 1

# Introduction

**Name: Bill**

- **B**eginner
- **I**ntermediate
- **L**earning
- **L**anguage

**Purpose:** It is a general purpose statically typed, with dynamic types, easy to learn and use language.

**Key uses:**

- Learning basic level programming.
- Pursuing enthusiast programming.

**Features:**

- portable
- a variety of useful data types
- extend-able data types
- simple to use
- constants
- ease of static typing
- strong static typing

Hello World

"Hello World" is traditionally the first program one writes in a new language. That makes it a good starting point.

```
#!/usr/bin/env bill

# hello.bill
# aka hello world

fun main():no_value
{
    writeln("Hello World")
    exit                    // defaults to 0
}
```

hello.bill

Of course, the first few lines are unnecessary. However, declaring a main function is required. For details, on that, in shapter 6 (Functions).

To compile and run:

```
prompt> bill hello.bill
```

If you operating system supports shebangs, the following will work:

```
prompt> hello.bill
```

## 1.1 Reserved Words

Here is a list of Reserved Words:

| Reserved Word | | See: |
|---|---|---|
| break | exit loop | Loops |
| continue | skip to next iteration | Loops |
| else | default condition | Loops |
| elsif | subsequent condition | Loops |
| exit | end program (possible exit value) | below. |
| if | condition | Loops |
| return | end a function (possible return value) | Functions |
| try | begin try block | Exceptions |

With exit, you can use any positive int64 value. However, 0 (default) generally represents a good run. Typically, a problem is designated by 1.

Next: Types

# Chapter 2

# Types

Types

Every form of data is a type. By extension, the same is true for functions. Built in types:

- null
- boolean
- int8
- int64
- float64
- string
- array
- vector
- set
- tuple
- dictionary

Types are based off of C++ types.

See https://en.cppreference.com/w/cpp/language/types

## 2.1 Integers

All integers are signed. See Unsigned for more information.

There are two integer types:

- int8 is 8 bit. (-128 to +127)
- int64 is 64 bit. (-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807)

*Tip:* Only use int8 for space conserving situations, provided all values will always fall within limits. Else, use int64. Don't trust "Ther's no reason for it to go beyond limits." It must be **impossible** to exceed limits or, it's only a matter of time.

### 2.1.1 Unsigned

Why no unsigned integers? Here is a great answer:

https://blog.robertelder.org/signed-or-unsigned-part-2/

## 2.2 Floats

The float 64 offers the same specs as c++ double.

## 2.3 Strings

```
#!/usr/bin/env bill

# numbers.bill

fun main():no_value
{
    var x:int8      = 9
    var y:int64     = 65536
    var z:float64   = 1 / 3

    var x2:int64    = int64(x)  // int8() and float64() work the same way
    var x$:string   = string(x)
    writeln(type(x) + " " + type(x2) + " " + type(x$))
}
```

numbers.bill

## 2.4 Boolean

TBD

## 2.5 Null

Use caution with null; abuse can be dangerous. See
https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/

## 2.6 Constants

The const reserverd word may be used to create a constant.

```
#!/usr/bin/env bill

# constant.bill

fun main():no_value
{
    const x:int8    = 9
    const y:int64   = 65536
    const z:float64 = 1 / 3

    const x2:int64  = int64(x)  // int8() and float64() work the same way
    const x$:string = string(x)
    writeln(type(x) + " " + type(x2) + " " + type(x$))
}
```

constant.bill

## 2.7 Others

For more information, please see Containers.

Next: Expressions

# Chapter 3

# Expressions

Expressions

Expressions work like most languages.

## 3.1 Math

```
#!/usr/bin/env bill

# expression.bill

fun main():no_value
{
    writeln(2 + 2)          // expression is 2 + 2
    writeln(4 * 2 + 3)      // evaluates 4 * 2 first
    writeln(4 * (2 + 3))    // evaluates 2 + 3 first

    writeln()               // Hint: prints a newline.

    var numerator:float64   = 1.0
    var denominator:float64 = 3.0
    var product:float64     = numerator / denominator
    writeln(product)

    writeln()               // Hint: prints a newline.

    // Integer division requires the floor division to work, which may not divide evenly.
    writeln(5 // 2)         // prints 2

    exit 0
}
```

expression.bill

### 3.1.1 Operators

| Urnary Operators | | Example |
|---|---|---|
| ++ | increment | i++ |
| – | decrement | i– |
| - | negative | -1 |
| **Binary Operators** | | **Example** |
| + | add | 2 + 2 |
| - | subtract | 4 - 2 |
| * | multiply | 4 * 2 |
| / | divide | 4.0 / 2.0 |
| // | floor divide | 4 // 2 |
| % | modulus | 4 % 2 |
| ** | power | 4**2 |
| **Inplace Operators** | | **Example** |
| += | add | x += 2 |
| -= | subtract | x -= 2 |
| *= | multiply | x *= 2 |
| /= | divide | x /= 2.0 |
| //= | floor divide | x //= 2 |
| %= | modulus | x %= 2 |

## 3.2 Logic

```
#!/usr/bin/env bill

# logic.bill

fun main():no_value
{
    var x:int8  = 4
    var y:int8  = 2

    writeln(x == y)         // prints false
    writeln(x > y)          // prints true

    exit 0
}
```

logic.bill

### 3.2.1 Operators

| Urnary Operators | | Example |
|---|---|---|
| ! | not | ! is_logical |
| ~ | invert | ~ x |
| **Binary Operators** | | **Example** |
| == | equal to | x == y |
| === | same identity | x === y |
| != | not equal to | x != y |
| < | less than | x < y |
| <= | less than or equal to | x <= y |
| > | greater than | x > y |
| >= | greater than or equal to | x >= y |
| in | membership | x in y |
| & | and | x & y |
| \| | or | x \| y |
| ^ | xor | x ^ y |
| << | shift left | x << y |
| >> | shift right | x >> y |

## 3.3 Concatenation

```
#!/usr/bin/env bill

# concatenate.bill

fun main():no_value
{
    var first_name:string   = John
    var last_name:string    = Doe
    var full_name:string    = first_name + " " + last_name
    writeln(full_name)

    writeln()               // Hint: prints a newline.

    // Direct concatenation of numbers is impossible. Use string().
    var age:int8            = 25
    writeln(full_name + " is " + string(age) + " years old.")

    exit 0
}
```

concatenate.bill

### 3.3.1 Operators

| Binary Operators | | Example |
|---|---|---|
| + | concatenate | word + " " |
| += | append | "Name: " += name |

Next: Containers

# Chapter 4

# Containers

Containers

Containers can hold multiple values.

## 4.1 Arrays

TBD

## 4.2 Vectors

Vectors, which are similar to lists, are sequences of data.

```
# vector.bill
# for vector samples

/*  @fn     main
 *  @brief  vectors
 */
fun main():no_value
{
    var primaries:vector
    primaries = ["red", "yellow", "blue"]
    writeln(primaries)          // prints ["red", "yellow", "blue"]

    var secondaries:vector = ["orange", "green", "purple"]
    writeln(primaries[1])       // prints green
    writeln(primaries[:2])      // prints ["orange", "green"]
    writeln(primaries[1:])      // prints ["green", "purple"]
    writeln(primaries[2:3])     // prints ["green", "purple"]
    // writeln(primaries[-1])       // prints purple

    var names:vector
    push(names, "Mandy")
    writeln(names)              // prints ["Mandy"]
    var names2 = tuple(names)   // works with sets too!
    writeln(names2)             // prints ("Mandy")
}
```

vector.bill

## 4.3 Sets

Sets are based on the mathenatical sets. See `sets` Note: sets are unordered. Accessing them will result in random ordering.

```
# set.bill
# for set samples

/*  @fn     main
 *  @brief  sets
 */
fun main():no_value
{
    var primaries:set
    primaries = {"red", "yellow", "blue"}
    writeln(primaries)      // prints {"red", "yellow", "blue"}

    var colors:set = {"red", "yellow"}

    // check subsets
    writeln(primaries < colors) // prints false
    writeln(colors < primaries) // prints true

    // check supersets
    writeln(primaries > colors) // prints true
    writeln(colors > primaries) // prints false

    // add more later

    // pop (unordered)
    write(pop(colors))          // prints either red or yellow (without a newline)
    write("\n")                 // prints a newline
    writelin(colors)            // prints whichever color was not popped
}
```

set.bill

### 4.3.1 Operators

| Binary Operators | | Example |
|---|---|---|
| < | subset | x < y |
| <= | subset or equal to | x <= y |
| > | superset | x > y |
| >= | superset or equal to | x >= y |
| in | membership | x in y |
| & | intersection | x & y |
| \| | union | x \| y |
| ^ | symmetric difference | x ^ y |

## 4.4 Tuples

```
# vector.bill
# for tuple samples

/*  @fn     main
 *  @brief  tuples
 */
fun main():no_value
{
    var primaries:tuple
    primaries = ("red", "yellow", "blue")
    writeln(primaries)      // prints ("red", "yellow", "blue")

    var secondaries:tuple = ("orange", "green", "purple")
    writeln(primaries[1])   // prints green
    writeln(primaries[:2])  // prints ("orange", "green")
    writeln(primaries[1:])  // prints ("green", "purple")
    writeln(primaries[2:3]) // prints ("green", "purple")
    // writeln(primaries[-1])  // prints purple
}
```

tuple.bill

## 4.5 Dictionaries

Note: dicts are unordered. Accessing them will result in random ordering.

```
# dict.bill
# for dictionary samples

/*  @fn     main
 *  @brief  vectors
 */
fun main():no_value
{
    var managers:dict
    managers = {"General": "Amy", "Assistant": "Bob", "Kitchen": "Tina"}
    writeln(managers)   // prints {"General": "Amy", "Assistant": "Bob", "Kitchen": "Tina"}
    writeln(managers["General])
}
```

dict.bill

## 4.6 Others

TBD

Next: Flow Control

# Chapter 5

# Flow Control

Flow Control

Flow control is about conditions.

## 5.1 Conditions

Condtions amount to Bool boolean states. E.G.:

- x > y
- i == 12
- 2 + 2 == 4
- fruit == "apple"
- etc (assuming etc is a boolean variable...)

Therfore the usual boolean rules apply here.

### 5.1.1 Conditionals

If

```
# if Conditional Example

fun main():no_value
{
    if true:
    {
        writeln(8)
    }

    // ternary expression
    var result:string
    result ? "Yes." : "No!"
    writeln(logical)
}
```

Else

```
# if - else Conditional Example

fun main():no_value
{
    if true:
    {
        writeln(8)
    }
    else:
    {
        writeln(2 + 3)
    }
}
```

Elsif

```
# Full Conditional Example

fun main():no_value
{
    if true:
    {
        writeln(8)
    }
    elsif false:
    {
        writeln(7, 9)
    }
    else:
    {
        writeln(2 + 3)
    }
}
```

### 5.1.2 Loops

While

```
# whileloop.bill
# while loop syntax sample

fun main(argsv):int8
{
    while true:
    {
        # This is the loop that never ends.
    }

    while 1 > 3:
    {
        # This loop is skipped.
    }

    var i:int8 = 0
    while i < argsv[1]:
    {
        writeln(i)
        i ++
    }
}
```

For

```
# forloop.bill
# for loop syntax sample

/*  @fn     main
 *  @brief  forloops
 */
fun main():no_value
{
    for var i:int64 = 0 to 10:
    {
        writeln(i)
    }

    for (var i:int64 = 0; i < 1):
    {
        writeln(i)
    }

    for (var i:float64 = 0; i < 1; i += .03):
    {
        writeln(i)
    }


    // foreach(variable, sequence)
    foreach(var primary:string, ["red", "yellow", "blue"]):
    {
        writeln(primary)
    }
}
```

## 5.2 Traversal

In the previos example, we slid in an example of travesing a vector. Traversing sets and tuples would follow the same pattern. Also, a container name could be substituted.

Dictionaries are a little trickier.

```
# dict.bill
# for dictionary samples

/*  @fn     main
 *  @brief  vectors
 */
fun main():no_value
{
    var managers:dict
    managers = {"General": "Amy", "Assistant": "Bob", "Kitchen": "Tina"}
    foreach(var key:string, keys(managers)):
    {
        writeln(managers[key])
    }
}
```

Next: Functions

# Chapter 6

# Functions

Functions

Encapsulating repeatable steps, is how we make programming easier.

## 6.1 Calling a Function

TBD

## 6.2 Declaring a Function

As seen in the Introduction, we have simple declarations.

```
#!/usr/bin/env bill

# hello.bill
# aka hello world

fun main():no_value
{
    writeln("Hello World")
    exit                    // defaults to 0
}
```

hello.bill

The funcion definition line should look familliar. The "fun" reserved word declares a function.

After the ":" is the function return type. However, in this case "no_value" indicates there is ho return of any kind.

A common practice of statically typed languages is to declare the type "void," which is similar.

## 6.3 Return

To return a value, use the resereved word return. See Nested (below)

Note: the return type must match the declared return type of the function. Otherwise a Static Error will occur, when attempting to compile.

## 6.4 Used in Expressions

Functions are easily added to expressions.

```
#!/usr/bin/env bill

# function_expression.bill

fun main():no_value
{
    writeln(cos(2))     // See cos in Standard Library Reference
    var value:float64   = sin(1 / 3)
    writeln(value)
    writeln("Amplitude: " + string(value))

    exit 0
}
```

function_expression.bill

## 6.5 Built-in

Here is a list of built-in functions:

| function | | See: |
|---|---|---|
| catch() | catch exception | Exceptions |
| float64() | convert to 64 bit float | Strings |
| int8() | convert to 8 bit integer | Strings |
| int64() | convert to 64 bit integer | Strings |
| keys() | return dictionary keys | Traversal |
| pop() | pop a value | Sets |
| push() | push a value | Vectors |
| string() | convert to string | Expressions |
| throw() | throw exception | Exceptions |
| tuple() | convert to tuple | Vectors |
| type() | get an object's type | Strings |
| write() | print (without newline) | Sets |
| writeln() | print (with newline) | Expressions |

## 6.6 Nested

Sometimes nested functions, limiting scope, may be useful.

```
#!/usr/bin/env bill

# nested-fun.bill

fun outer():int8
{
    fun inner(x:int8, y:int8):int8
    {
        return x + y
    }

    return inner(2, 3)
}

fun main():no_value
{
    writeln(outer())
    exit 0
}
```

nested_fun.bill

Next: Exceptions

# Chapter 7

# Exceptions

Exceptions

As we all know, "Things don't always go according to plan." Hence programmers need to account for this, with exception handling.

## 7.1 Throw

Let's just throw this out.

```
# Throw!

fun main():no_value
{
    throw("Something happened!")
}
```

## 7.2 Try

First a simple example.

```
# Try something!

fun main():no_value
{
    try:
    {
        writeln(8)
    }
    catch():
    {
        throw("What happened?")
    }
}
```

This is ok, if there is no concern over "What went wrong?"

## 7.3 Catch

Now, let's catch the exception.

```
# Try...catch!

fun main():no_value
{
    try:
    {
        writeln(8)
    }
    catch(exception):
    {
        throw(exception + " happened!")
    }
}
```

However, this only catches a specific exception.

```
# Try...indexError!

fun main():no_value
{
    try:
    {
        writeln(8)
    }
    catch(indexError):                  // Specific exception caught.
    {
        throw("Index out of range.")    // Specific exception handled.
    }
    catch(exception):                   // Unknown exception caught.
    {
        throw(exception + " happened!") // Unknown exception handled.
    }
}
```

By daisy chaining catches, we can fine tune the response.

Next: Style

# Chapter 8

# Style

Style

Code style can be a matter of choice...

However, consistency means readability. As such, here are coventions used throught this documentation.

## 8.1 Comments

Possible comment types:

```
#   This is a comment type recommended for shebangs.
//  This is the recommended end-of-line comment.
/*  This type of comment is recommended for documentation blocks. */

//  or

/*  myfunction
 *  Demo an operation
 */
```

## 8.2 Statements

TBD

## 8.3 Blocks

```
Declaration:    // if, while, etc...
{
    // Code here.
}
```

# The Pragmatic Language
## Language Primer

Bill is a simple, general purpose, statically typed language. It's good for the static novice. Easy to learn and use, the langage is quite versitile.

This document is designed for those who already have some knowlege of programming. Complete with code samles, demonstrating how to use the syntax.