The Pragmatic Language

Language Primer

This document is disigned to help progammers learn the basics of Bill.

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:

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

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

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)

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

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 Boolean

2.4 Strings

2.5 Null

2.6 Others

For more information, please see Containers.

Next: Expressions

Chapter 3

Expressions

Expressions

Express!

3.1 Math

3.2 Concatenation

Next: Containers

Chapter 4

Containers

Containers

Containers can hold multiple values.

4.1 Arrays

4.2 Vectors

4.3 Sets

4.4 Tuples

4.5 Dictionaries

4.6 Others

Next: Flow Control 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)
    }
}
```

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  simple forloop
 */
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

Next: ref Functions

Chapter 6

Functions

Functions

Encapsulating repeatable steps, is how we make programming easier.

6.1 Calling a Function

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

6.4 Used in Expressions

6.5 Built-in

Here is a list of built-in functions:

catch()	catch exception
throw()	throw exception
toFloat()	convert to float
toInt()	convert to integer
toString()	convert to string
toTuple()	convert to tuple
type()	get an object's type
write()	print (without newline)
writeln()	print (with newline)

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, heare 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

8.3 Blocks

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