

# The Cross Application Framework

---

**Current Revision Version:** 1.0.1

## Document Revision History

Version	Author	Date	Revision
1.0.0	Michael Schutt	06/13/2019	Initial documentation content creation.
1.0.1	Michael Schutt	06/19/2019	Added Lexical and Syntactic Grammar section, tutorial examples and hyperlinked sections with table of contents.
1.0.2	Michael Schutt	06/20/2019	Added Shell documenation section, correct grammar and misspellings.

## Table of Contents

---

- **Introduction**
  - [Background on the Author](#)
  - [The Motivation Behind Cross](#)
  - [The Multifaceted Meaning Behind the Name](#)
  - [Formatting Conventions Used](#)
- **The Cross Application Framework**
  - [The Cross Programming Language](#)
    - [Overview](#)
    - [Design Principles](#)
    - [Influences](#)
    - [Language Specification](#)
      - [Lexical Grammar](#)
        - [Source Files](#)
          - [Encoding](#)
          - [Empty Modules](#)
        - [Whitespace](#)
          - [Space Characters](#)
          - [Indentation Characters](#)
          - [Newline Characters](#)

- Identifiers
- Literals
  - Boolean Literals
  - Numeric Literals
  - String Literals
  - Aggregates Data Type Literals
- Keywords
  - The `use` Keyword
  - The `log` Keyword
- Operators
  - Arithmetic Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
    - Self-Assignment Operators
- Syntactic Grammar
  - Modules
    - Module Naming
    - Module Loading
  - Expressions
    - Arithmetic Expressions
    - Logical Expressions
    - Bitwise Expressions
  - Statements
    - The Assignment Statement
    - The `use` Statement
    - The `log` Statement
    - The `next` Statement
    - The `exit` Statement
    - The `throw` Statement
    - The `return` Statement
  - Blocks
    - The `if` Block
      - The `elif` Block
      - The `else` Block
    - The `while` Block
    - The `for` Block

- The `case` Block
  - The `of` Block
- The `try` Block
  - The `on` Block
  - The `then` Block
- The `fun` Block
- The `class` Block
- Cross Command-Line Interactive Compiler Shell
  - Command-Line Flags
    - The `-d/--debug` Flag
    - The `-v/--version` Flag
    - The `-l/--license` Flag
  - Interactive Commands
    - The `.clear` Command
    - The `.debug` Command
    - The `.editor` Command
    - The `.reset` Command
    - The `.break` Command
- Cross Code Editor Integration
  - Supported Code Editor Applications
    - Microsoft Visual Studio Code
      - Official VSCode Marketplace Extensions
        - Cross Syntax Highlighting
          - Realtime Syntax Highlighting
          - Support for `.cross` File Type
      - Editor Configuration for Cross Development
        - Hosting Cross Interactive Shell within Visual Studio Code Integrated Terminal
- Cross Optimizing Compiler
  - Overview
  - Performance
  - Optimizations
  - Execution Engine
- CrossPack Package Manager
  - Installation
  - Configuration
  - Package Management

- Installing Packages
- Uninstalling Packages
- Restoring Packages
- Creating Packages
- Publishing Packages
- Removing Packages

- **Frequently Asked Questions**
- **Tutorials**

## Introduction

---

This document is both the official specification for the Cross Application Framework and a useful guided tour of the language and tools Cross provides. The lexical and syntactic grammars are defined in detail as well as the various types, keywords, control constructs and syntax. The following explains the motivation, purpose, language constructs and tools that are used when developing using the Cross Application Framework.

## Background on the Author

The Cross Application Framework, language design and implementation was accomplished by Michael Schutt (whose online moniker is @crosscripser). A professional software developer with nearly two decades of real world industry experience, passionate about development and the pursuit of maximizing developer productivity, set out to develop a unique set of symbolic shorthand characters which would represent commonly used HTML elements in order to speed up the development of HTML web pages.

## The Motivation Behind Cross

This initial objective was quickly realized and other requirements such as the ability to embed expressions and to repeat an element a given number of times became apparent, which ultimately forced Michael to add expression parsing and evaluation to his small shorthand HTML compiler. Eventually the full fruition of the small seed of creativity to simply speed up the development of HTML began to grow and produce new fruitful ideas on how to genericize the initial core idea, only this time applying the same minimalistic development principles to all of coding and thus the realization of a requirement for a proper, fully featured, turing complete programming language was at last in motion within his mind. The initial short hand symbols of C.R.O.S.S (or Characters Representing Objects Scripting Syntax), language would not work when adding flow control blocks, expressions, type systems and other language features.

He soon realized that a new minimalistic, yet fully expressive syntax must be designed for writing a full Cross programming language. Michael drew from his years of experience diving into the vast sea of programming languages available, to pull out the best and most productive ideas from major languages and unify them into a single perfect syntax. Spending years distilling programming language syntax, constructs and features down to the core ideas, he managed to boil it all down to a minimal subset of these important and necessary requirements. But the original vision was (and still is) to ultimately distil *development itself* down to it's core and provide a minimal set of language syntax and build tools which yield expressive, readable and performant code. We think the official tagline says it all, "Code Less, Develop More™".

While removing the language design flaws and unnecessary traditional punctuation inherited from the languages of the past, like semicolons at the end of new lines, a vestige left over from the punch card days, or curly braces as block delimiters, inherited from the C-Family tree.

He realized these all too familiar syntactic comforts were not *strictly necessary* and thus should be eliminated from a language syntax that strove towards minimalism and expressivity. He came to the conclusion that semicolons could be fully replaced by simply a newline character to demarcate the end of a given line of code. Instead of curly braces to delimit where blocks begin and end, why not use the whitespace indentation level alone, which is almost already typed anyway even in braced languages instead of by explicit begin and end markers such as curly braces or keywords like in other languages.

But he went further in the redesign of programming languages. If code was to be as minimal as possible then all code must be short, clear and concise. Therefore language keywords came under fire next, realizing that some of the most popular languages seem to have lengthy keywords such as `continue`, `function`, `interface`, and `return` he vowed to only include the *shortest* therefore *most easily typed* keywords in place of traditional keywords. Because of this we find short, expressive keywords such as `use` instead of `require` or `imports`, `Use` is a simple, short and conveys exactly what we intend to do, namely `use` a module.

Other examples of this kind of replacement are the use of `elif` for `else if`, the use of `case` for `switch` and `of` for the traditional `case`, `then` for `finally`, `next` for `continue` and `as` for `extends` or `inherits` and so on.

The language syntax started to resemble a smaller minimal subset of written code, but the problem of types suddenly came to light, In order to remove all explicit type names, Michael decided to make all types implicit and depend on the compiler to infer the types. By making them static and strongly typed they would still retain their reliability in the code base, yet the burden of the type names was on the compiler not the programmer. This gives the impression of a light, dynamic like syntax but with the robust type-safe security and compile time error checking of a traditional strong static typed language, this unique blend of the two typing methodologies he dubbed *Implicit Strong Static Typing*.

Once the keywords were dropped or shortened, types were inferred and unnecessary punctuation was removed, there was surprisingly not much left of written code except for the user's naming conventions and a small bit of expression and statement separator syntax. Michael then turned his attention towards refactorability. Code bases change, all the time but sometimes quite drastically and he wanted to avoid the hassle of renaming a definition in code in multiple places. One such notorious place is the class constructor of a traditional C-family language. It usually will require a class's name to match the constructor name exactly. Change a class's name and you have to remember to change all of it's potential constructor overloads. Michael decided in order to keep the implicit and minimal approach to his syntax, constructors were inlined right after the name of the class that way the class name and the constructor name are in one place, Rename or refactor one and you automatically have done the other. Might not seem like much of a change, but this saves time in development, refactoring and copy and pasting code where it's not necessary to do so. Refactoring should be the compiler's task not the developer's. Features such as inline constructors simplified the refactoring process which in turn speeds up the development over time and reduces the mental payload required to make successful changes to an existing code base. Maintainable code is key.

With all of these small changes and a couple of new innovative language ideas such as the concept of implicit imports, iteration and inheritance -- the new goal was to make code accessible. Everyone no matter what language they write in, nor what platform they code on, should be able to write clean, simple, code. So Michael realized that programming was not just an American English task, No, millions of multi-lingual developers are writing code each and every day. In thousands of different languages. A modern language

would need to support a global audience, therefore Cross has full Unicode UTF8 encoding support for modules as this was needed for a multi lingual programming language. But yet another boundry exists between developers and that is one of computing platforms, including a diverse selection of major operating systems such as Microsoft Windows, Apple's MacOS and thousands of differing GNU Linux distributions. If Cross was going to be used by such a diverse user base as this, it would not only need to be multilingual, but also *cross-platform* running on any major operating system a developer may have on their machine.

## The Multifaceted Meaning Behind the Name

### Professional Meaning

This made Michael realize that the tool he was building was not just *cross-platform* but it was truly *cross-everything*, it was *cross-cultural*, *cross-platform*, and *cross-paradigm*. This began to shed a new light on the original name of the language which initially started out as a simple acronym, but had now somehow become the embodiment of a concept, of a universal, *cross-everything* language...in it's own right, representing the crossing of traditional boundries which siloed developers from each other. Cross was starting to take on a new meaning and Michael adopted it whole heartily.

Through it all Michael has had a guiding quote which he abides by as the foundation from which he builds all language and framework features, a simple but profound inspiring quote by Antoine de Saint-Exupery a french novelist:

"Perfection is achieved  
not when there is nothing more to *add*  
but when there is nothing left to *take away*."

### Spiritual Meaning

Without a doubt, the most important and underlying symbolic meaning in the name of Cross is that of the Christian Cross, Michael being a born again Christian attributes all his ability, skills and knowledge to his Lord Jesus Christ and the power of His Cross, which radically changed the direction of Michael's life during his teenage years. Therefore, the entire project and any glory that may come from it is therefore dedicated to God and His Son Jesus Christ who by the power of the Cross has delivered all who will believe in Him from the power of death, hell and the grave.

Michael prays that the power of his Cross might have such a global and far reaching impact on development and that it would be ultimately used to help solve complex and real world problems and to better our world.

## Formatting Conventions Used

---

Various formatting conventions are used throughout this document to represent the differences between text types, for example input vs output text from the shell etc. The following is a list of textual formatting conventions used to convey different contexts in this document:

### Source Code

Source code is shown using the bordered box around the text, there are two different types of source code convention, one to show short inline code snippets and the other is reserved from multiple lines of example

code:

Inline code is shown like this

and

```
Blocks of source  
code are shown  
in this style
```

## The Cross Application Framework

---

The Cross Application Framework consists of a set of development tools to help build modern software applications. The toolset includes the Cross command-line interactive compiler shell, the Cross programming language and compiler and the CrossPack package manager system.

### The Cross Programming Language

#### Overview

The Cross programming language is the core of the Cross Application Framework. It is the modern, high-level, multi-paradigm, general purpose programming language.

#### Design Principles

It focuses on minimalism and conciseness while still maintaining expressiveness and clarity. It's both readable and writable. It's multi platform, multi lingual and multi paradigm, supporting any style of development, in any language, on any platform.

#### Influences

Cross takes influence from many different popular programming languages, and generally speaking the best ideas from each language were taken into consideration, such as whitespace block delimiting, expression chaining, in keyword and more from Python, the class, base and OOP and typing concepts from C#, the list and map literals from Javascript and much more. Each language I've experienced has helped form and polish the Cross language syntax into a collective, minimal but expressive language.

## Language Specification

---

### Lexical Grammar

The Lexical grammar is used by the lexical scanner to determine how to group together sets of individual characters to form tokens, which are small units of characters tagged with a type and a file position. The lexical grammar defines all lexical syntax, in that it provides definitions for what are valid literals, keywords, separators and operators combinations. The following grammar will sometimes use EBNF notation for the various production definitions.

## Source Files

A unit of Cross source code is called a *module*, that is a physical text file with the extension .cross which contains executable/compilable Cross source code as input.

### Encoding

Each source file, called in Cross a module, is expected to be encoded in UTF-8 (without the Byte Order Mark).

### Empty Modules

An completely empty module that contains no source code is an error, a module must contain at least one valid lexical element. Whitespace and newlines count as such an element.

## Whitespace

Whitespace is any of the following characters listed below, these characters are used throughout the module to separate both syntactic and lexical elements.

### Space Characters

The following Unicode character codes are valid whitespace elements:

- \u0020 - Space

### Indentation Characters

The following Unicode character codes are valid indentation elements:

- \u0020 - Space
- \u0009 - Horizontal Tab \t

NOTE: Tabs and spaces can be mixed freely, the Cross Normalizer will normalize a single Tab element into the equivalent four (4) Space elements

### Newline Characters

The following Unicode character codes are valid newline characters:

- \u0010 - Carriage Return (\r)
- \u0013 - Newline (\n)

NOTE: Different platforms use different combinations of these codes to represent starting a new text line, for example:

- Windows uses: \u0010\u0013 (\r\n)
- MacOS uses: \u0010 (\r)
- Linux uses: \u0013 (\n)

These newline character sequences are normalized by the Cross Normalizer into the equivalent of a one (1) \u0013 (\n) standard newline element



## Identifiers

Identifiers in Cross following standard identifier naming conventions used in most other programming languages. A valid identifier token must:

- Start with at least one (1) valid start element, one of: underscore (`_`) or an case-insensitive alpha letter (a-z)
- Continue with a valid start element and/or a digit (0-9)

Examples of good, valid identifiers:

- `debug`
- `__privateKey`
- `ConfigurationManager`
- `windows10`
- `__SUPER_PRIVATE_ID__`

Examples of invalid identifiers:

- `6dozenEggs` - *cannot start with a digit*
- `$jQueryRocks` - *cannot start with reserved start element \$*
- `!important` - *symbols are not allowed as start or continuation elements except for underscore (`_`)*
- `first-name` - *hyphens are not allowed as start or continuation elements*

NOTE: Unicode characters of the class L (`\p{L}`) **are** allowed in identifiers as valid start and continuation elements:

Examples of valid Unicode identifiers:

- `שנס`
- `اسمي`
- `Alefא`
- `__Shinش`
- `V4YİİþΓΞñæmé`

## Literals

Cross includes literal syntax for all primitive data types. A literal must be one of the following types:

### Boolean Literals

Type	Syntax	Data Represented
<code>bool</code>	<code>true</code> , <code>false</code>	Logical values

### Numeric Literals

Type	Syntax	Data Represented
<code>number</code>	<code>1</code> , <code>1.0</code> , <code>-32</code> , <code>3.1415926</code>	Numeric values

## String Literals

Type	Syntax	Data Represented
<code>string</code>	<code>'abc', "def"</code>	Textual values

## Aggregate Data Type Literals

In addition to the three (3) primitive data types, there are aggregate data types which group primitive values together.

The aggregate data type literals must be one of the following types:

Type	Syntax	Data Represented
<code>list</code>	<code>[], [1], ['a', 'b']</code>	Multiple values
<code>map</code>	<code>{}, {'x': 1}, {x:1, y:'2'}</code>	Sets of keys and values

## Keywords

In Cross there are as few keyword as possible, each keyword has been specifically chosen for its descriptiveness and brevity.

The following is a list of **all** reserved keywords in the Cross programming language:

```
and as base case class elif else exit
false for fun if in next not of on
or return then throw true try use while
```

## The `use` Keyword

A good example of this is the keyword `use`, while other languages use the keywords `imports` or `require` or `using` which all describe importing a module, Cross chose a short, descriptive keyword instead `use`:

```
# Imports the module module
use math
```

## The `log` Keyword

The `log` keyword is actually **technically** not a keyword, however it's highlighted as such and is used so often it could be thought of almost a keyword, it's the `io::log` function from the built in system module `io`. The `log` keyword is used to **log** information to the system console (ie. usually stdout).

```
# Print that infamous message to the screen.
log('Hello, World!') # Hello, World
```

The `log` function can take any expression, but usually takes a `string` expression to print. However, any expression is automatically cast to a `string` and printed when passed as an argument:

```
log(3.1415926) # 3.1415926
log(true)      # true
log([1, 2, 3]) # [1, 2, 3]
```

## Operators

Operators in Cross are symbols that perform mathematic, logical or bitwise operations on their operands. Operators can be any *fixity*, meaning their position within their operands, one of infix, prefix or postfix.

- Prefix: Comes before the right-hand operand
- Infix: Comes between the left and right-hand operands
- Postfix: Comes after the left-hand operand

NOTE: Some operators like Addition (+) or Subtraction (-) can have **multiple** fixities! For example:

- `+1` - *prefix*
- `1 + 1` - *infix*
- `1++` - *postfix*

## Arithmetic Operators

Cross contains all the usual arithmetic operators for adding, subtracting, multiplication, division etc. Below is a full table of the available arithmetic operators:

Operator	Fixity	Syntax	Operation
Positive	Prefix	<code>+a</code>	Returns the positive numeric value for the right-hand operand
Negation	Prefix	<code>-a</code>	Returns the negative numeric value for the right-hand operand
Addition	Infix	<code>a + b</code>	Adds the left and right-hand operands together
Subtraction	Infix	<code>a - b</code>	Subtracts the left-hand operand from the right-hand operand
Multiplication	Infix	<code>a * b</code>	Multiplies the left-hand operand by the right-hand operand
Division	Infix	<code>a / b</code>	Divides the left-hand operand by the right-hand operand
Modulo	Infix	<code>a % b</code>	Returns the integer remainder of the division of the left-hand operand by the right-hand operand

## Logical Operators

Cross supports logical and comparison operators for testing for truthiness. Below is a full table of the available logical operators:

Operator	Fixity	Syntax	Operation
----------	--------	--------	-----------

Operator	Fixity	Syntax	Operation
Less Than	Infix	$a < b$	Returns <b>true</b> if the left-hand operand is less than the right-hand operand
Greater Than	Infix	$a > b$	Returns <b>true</b> if the left-hand operand is greater than the right-hand operand
Less Than or Equal to	Infix	$a \leq b$	Returns <b>true</b> if the left-hand operand is less than <b>or</b> equal to the right-hand operand
Greater Than or Equal to	Infix	$a \geq b$	Returns <b>true</b> if the left-hand operand is greater than <b>or</b> equal to the right-hand operand
Equality	Infix	$a == b$	Returns <b>true</b> if the left-hand operand is equal to the right-hand operand
Inequality	Infix	$a != b$	Returns <b>true</b> if the left-hand operand is <b>not</b> equal to the right-hand operand
Logical Not	Prefix	$!a$	Returns the logical negation of the right-hand operand where: <b>!true</b> evaluates to <b>false</b> and <b>!false</b> evaluates to <b>true</b>

## Bitwise Operators

Cross also supports bit/byte binary-level operations using the bitwise operators. Below is a full table of the available bitwise operators:

### Bitwise Combination Operators

Operator	Fixity	Syntax	Operation
Bitwise AND	Infix	$a \& b$	Returns the bitwise AND of the left-hand operand and the right-hand operand
Bitwise OR	Infix	$a   b$	Returns the bitwise OR of the left-hand operand and the right-hand operand
Bitwise XOR	Infix	$a \wedge b$	Returns the bitwise XOR of the left-hand operand and the right-hand operand
Bitwise NOT	Prefix	$\sim a$	Returns the bitwise negation of the right-hand operand where: <b>0</b> evaluates to <b>1</b> and <b>1</b> evaluates to <b>0</b>

Cross also supports bitwise shift operations, left and right variations: **Bitwise Shift Operators**

Operator	Fixity	Syntax	Operation
Bitwise Shift Left	Infix	$a \ll b$	Returns the bitwise SHL of the left-hand operand and the right-hand operand
Bitwise Shift Right	Infix	$a \gg b$	Returns the bitwise SHR of the left-hand operand and the right-hand operand

## Bitwise Rotate Operators

Operator	Fixity	Syntax	Operation
Bitwise Rotate Left	Infix	a <<< b	Returns the bitwise ROTL of the left-hand operand and the right-hand operand
Bitwise Rotate Right	Infix	a >>> b	Returns the bitwise ROTR of the left-hand operand and the right-hand operand

## Assignment Operators

Cross allows assignments to be completed to variables themselves or other variables using the assignment operators listed below:

### Assignment Operators

Operator	Fixity	Syntax	Operation
Assignment	Infix	a = b	Assigns the value of the right-hand operand to the storage on the left-hand operand

## Self-Assignment Operators

Cross also supports a shorthand syntax for assigning a express back to the variable itself, called self-assignment operators. Most operators support the shorthand self-assignment syntax, though not all. The following is a list of supported self-assignment operators:

### Self-Assignment Arithmetic Operators

Operator	Fixity	Syntax	Operation
Self Addition	Infix	a += b	Assigns the value of the addition operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Subtraction	Infix	a -= b	Assigns the value of the subtraction operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Multiplication	Infix	a *= b	Assigns the value of the multiplication operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Division	Infix	a /= b	Assigns the value of the division operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Modulo	Infix	a %= b	Assigns the value of the modulo operation on the left-hand operand and right-hand operand back to the left-hand operand.

### Self-Assignment Bitwise Combination Operators

Operator	Fixity	Syntax	Operation
----------	--------	--------	-----------

Operator	Fixity	Syntax	Operation
Self Bitwise AND	Infix	$a \&= b$	Assigns the value of the bitwise AND operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Bitwise OR	Infix	$a \mid = b$	Assigns the value of the bitwise OR operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Bitwise XOR	Infix	$a \wedge = b$	Assigns the value of the bitwise XOR operation on the left-hand operand and right-hand operand back to the left-hand operand.

### Self-Assignment Bitwise Shift Operators

Operator	Fixity	Syntax	Operation
Self Bitwise Shift Left	Infix	$a \ll = b$	Assigns the value of the bitwise SHL operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Bitwise Shift Right	Infix	$a \gg = b$	Assigns the value of the bitwise SHR operation on the left-hand operand and right-hand operand back to the left-hand operand.

### Self-Assignment Bitwise Rotate Operators

Operator	Fixity	Syntax	Operation
Self Bitwise Rotate Left	Infix	$a \lll = b$	Assigns the value of the bitwise ROTL operation on the left-hand operand and right-hand operand back to the left-hand operand.
Self Bitwise Rotate Right	Infix	$a \ggg = b$	Assigns the value of the bitwise ROTR operation on the left-hand operand and right-hand operand back to the left-hand operand.

## Syntactic Grammar

The *syntactic grammar* is used by the parser to determine how to group together sets of tokens to form syntax trees, which are linked nodes of tokens with extra type, scope and a file position information. The syntactic grammar defines all expression, statement and block level syntax, in that it provides definitions for what are valid expressions, statements, and blocks. The following grammar will sometimes use EBNF notation for the various production definitions.

### Expressions

Cross expressions are primitive data type values, subexpressions, or combination of operators and primitive data type values as operands. Expressions can one of the following types:

#### Arithmetic Expressions

Arithmetic expressions consist of `number` literals or variables of the `number` type, or a combination of arithmetic operators and numeric operands.

## Logical Expressions

Cross logical expressions are used as conditions and expression values to assignments. Conditions are passed to flow control blocks in order to affect program flow. Logical expressions contains `boolean` literals, `boolean` subexpressions or comparisons which yield `boolean` values, or a combination of these element as operands to a `logical operator`

```
true
false
!true           # Negated expressions
1 < 2           # Comparisons
true or false   # Boolean expressions
!false and 3 > 2.0 # Combination of all
```

## Bitwise Expressions

Cross bitwise expressions are made up of integer values, or a combination of bitwise operators and integer operands.

```
1 & 2           # AND
1 | 2           # OR
1 ^ 2           # XOR
~0              # NOT
~1 >> 2         # Shift right
2 <<< 4         # Rotate left
~1 << 2 & 3     # Combination of all
```

## Literal Expressions

Cross literal expressions are expressions made up of `number`, `boolean` or `string` literals, or a aggregate data type literal made up of those values.

### Boolean Literals

`boolean` literals consist of the constant keyword values `true` and `false`:

```
true
false
```

### Numeric Literals

**number** literals can be of integer form (fixed) or floating point form. Numeric literals represent quantifiable, numeric values which uses double-precision floating-point format numbers as specified in IEEE 754 and can only safely represent numbers.

### Integer Expressions

**int** (short for **integer**) literals can be of fixed form. Integer expressions are single, fixed (non floating point) numeric literals which consist of values ranging from -9,007,199,254,740,991 to +9,007,199,254,740,991:

```
1
-2
32
1000000
```

### Floating Point Expressions

**float** literals are of the floating point form, Float expressions are single, floating point numeric literals which consist of values ranging from approximately 5e-324 to 1.79E+308 (or  $2^{1024}$ ):

```
1.0
-2.0
0.3456
0.00004
+5.00000
```

### String Literals

String literals are textual values delimited by either single quotes (') or double quotes ("). Strings can contain any character, including the same quote which delimits it by escaping the quote using the escape backslash (\) before the delimiting quote (" or '). The escape backslash (\) can also be used to insert special *escape sequences* which represent special characters which are hard or impossible to type on a keyboard:

### Escape Sequences

Name	Sequence
Alert	\a
Backspace	\b
Form Feed	\f
Newline	\n
Carriage Return	\r
Horizontal Tab	\t



Name	Sequence
Vertical Tab	<code>\v</code>

```
"abc"           # Double quote string
'def'           # Single quote string
"This is a string"
'This is also a string'
"st\r\n"        # Escaped strings

# Multiline string
"This is
also a string
on multiple
lines"
```

NOTE: Unlike most other programming languages, Cross **allows** newlines in `string` literals. No triple quotes, no `@` before the `string` just write your string across multiple lines!

## String Interpolation

In Cross expressions or values can be *interpolated* or place directly into a `string` literal by using the *interpolation operator* (`$`):

```
name = 'Mike'
age = 33

"$name is $age"    # Interpolated values
"$'hello' world"   # Interpolated strings
"1 + 2 = $(1 + 2)" # Wrap spaced expressions with parens
"$'abc'.length"    # Interpolation properties
```

## Statements

### The Assignment Statement

```
name = 'Mike'
age = 33
gpa = 4.0 # yeah right
married = true
smoker = false
kids = ['Micaiah']
contact = {email: 'crosscripiter@gmail.com'}
```

### The use Statement

```
# Import the math module
use math
```

## The log Statement

```
log('Hello, World!')
```

## The next Statement

```
for x in [1, 2, 3]
  if x == 2
    next    # skip 2
  log(x)
```

## The exit Statement

```
for x in [1, 2, 3]
  if x == 2
    exit    # break out of loop
  log(x)
```

## The throw Statement

```
# Throw any expression
throw 'Ahhh, Real Monsters!'
```

## The return Statement

```
fun add(a, b)
  return a + b    # returns the sum of a and b
```

## Blocks

Blocks in Cross are used to control the flow of the program. All the usual blocks from most popular programming languages are available. Cross adds some unique keywords to some of the familiar ones however. Instead of `switch ... case ... default` Cross instead uses the `case ... of ... else` block for example, or instead of `try ... catch ... finally`, instead `try ... on ... then`, simpler, shorter and still as descriptive keywords:

### Single Line Block Shorthand

In Cross if your block consists of a single statement you can use the shorthand single line block delimiter (:), for example:

```
if 1 < 2
    log('less than')
```

Shorthand single line block delimiter syntax:

```
if 1 < 2: log('less than')
```

### The if Block

```
if 1 < 2
    log('less than')
```

or using single line block syntax:

```
if 1 < 2: log('less than')
```

### The elif Block

```
if 1 < 2
    log('less than')
```

```
elif 1 > 2
    log('greater than')
```

### The else Block

```
if 1 < 2
    log('less than')
elif 1 > 2
    log('greater than')
```

```
else
    log('equal to')
```

NOTE: All blocks and subblocks (like `elif` and `else`) can use the single line block syntax shorthand too, for example:

```
if 1 < 2: log('less than')
elif 1 > 2: log('greater than')
else: log('equal to')
```

## The `while` Block

```
# Countdown using while
x = 10

while x > 0
    log('$x...') # 10...9...8...
    x -= 1

log('BLASTOFF!')
```

## The `for` Block

```
# Iteration through lists
fruits = ['apples', 'bananas', 'oranges']

for fruit in fruits
    log('I love $fruit!')
```

```
# Iteration through maps
friends = {'Bob': 52, 'Mary': 43, 'Jim': 75, 'Jack': 10}

for name, age in friends
    log('$name is $age')
```

## The `case` Block

```
day = "Fri"

case day
of "Mon"
of "Tue", "Wed", "Thu"
    log("Work")
of "Fri": log("Party")
```

```
of "Sat"
  log("Yard work")
else
  log("Church?")
```

### The of Block

```
age = 16

case age
of < 12: play()
of >= 13 and <= 15: hangout()
of 16: drive()
of >= 21: drink()
of >= 65: retire()
else: bingo()
```

### The try Block

```
try
  connect()
on e
  throw 'Could not connect: Error was $e'
then
  log('Connection complete')
```

### The on Block

```
try
  log(0 / 0)
on e as DivideByZeroError
  Universe.implode()
on regularOleError: throw
```

### The then Block

```
try
  file.delete()
on err: throw
then: log('File deleted?')
```

### The fun Block

```
fun hello
  log('Hello')

hello()
```

```
fun greet(name)
  log('Hello, $name!')

greet("World")
```

```
fun add(a, b) => a + b
log(add(1, 2))
```

## Lambdas

```
# lambda syntax
hello = () => log('Hello')
greet = name => log('Hello, $name!')
add = (a, b) => a + b
```

## The class Block

```
class Thing
```

```
class Box(contents)
shoebox = Box('shoes')
```

```
class Point(x, y)
  fun coords => [x, y]

class Vector(x, y, z) as Point(x, y)
  fun coords => [...base.coords(), z]







p = Point(1, 2)
v = Vector(-1, 0, 1)
points = [p, v]
```

```
for point in points
  log(point.coords())
```

## Cross Command-line Interactive Compiler Shell

The Cross command-line interactive compiler shell, or simply the Cross shell, is a repl (Read Evaluate Print Loop) interface into the Cross compiler. It's interactive in that it accepts input and gives some type of output for each line entered, whether that be the output or an error message. The Shell also has a system of prompts which indicate in which mode the repl is currently in, which will be one of the following:

### Shell Prompts

Prompt	Mode	Description
	Input	Shell is waiting for the next line of user input to be entered and the <b>ENTER</b> key to be pressed
	More	Shell is waiting for a multi-line user input to be terminated (eg. multiline strings, literals, blocks etc.)
	Output	Shell is displaying the result of the evaluation of the last user input entered
	Error	Shell is displaying the error message and position of the error thrown during the evaluation of the last user input line
	Log	Shell is displaying debugging or other logging information to the current user
	Help	Shell is waiting for user help term to be entered and the <b>ENTER</b> key to be pressed to begin help search.

A typical shell session would look something like this:

```
>> x = 1
=> 1
>> if x == 1
..   log(true)
..
=> true
>>
>> x = 1 +
!> repl(1, 8) Syntax Error: ...
>> .debug
$> [06/20/2019 12:13:00 AM] Debug mode enabled
?> string
```

## Command-line Flags

The Cross Shell accepts command-line flags to enable or disable certain features. The command-line flags for the Cross shell must be one of the following:

### Command-Line Flags

Flag (short/long)	Name	Description
<code>-d / --debug</code>	Debug	Enables debug mode, which logs detailed information from the compilation process.
<code>-v / --version</code>	Version	Displays the full version number of the Cross Application Framework used by the current shell.
<code>-l / --license</code>	License	Displays the full license key and registered user information for the Cross Application Framework used by the current shell.

The Cross Shell also has a built-in command system. These commands affect the Shell in different ways, enabled and disabling certain features or options. To use a built-in command, start with dot (.) and then the command name to distinguish a command from code. The built in shell commands are the following:

### Built-in Shell Commands

Command	Name	Description
<code>.debug</code>	Debug	Toggles debug mode on and off interactively during the shell session.
<code>.clear</code>	Clear	Clears the shell screen and returns to input mode.
<code>.editor</code>	Editor	Enters multi-line editing mode, allows multiple lines of input to be entered at once.
<code>.break</code>	Break	Breaks out of a multi-line editing session or input and returns to a new input mode.
<code>.reset</code>	Reset	Resets the shell input/output state and starts a new repl session
<code>.help</code>	Help	Starts the interactive help mode within the current shell session
<code>.exit</code>	Exit	Exits the current shell session and defers input/output back to the host system shell.

## FAQs (Frequently Asked Questions)

## Tutorials

### hello.cross

```
# Print the infamous hello world message to the console
log('Hello, World!')
```

### shapes.cross



```

# Import math module for pi function
use math

# Define a base Shape class
class Shape

# Define a Rectangle shape class
class Rectangle(width, height) as Shape
  fun area => width * height

# Define a specialized subclass Square as Rectangle with equal sides
class Square(size) as Rectangle(size, size)

# Define a Circle subclass that implements area differently
class Circle(radius) as Shape
  fun area => math.pi() * radius ** 2

# Create some new Shapes
shapes = [
  Rectangle(2, 4),
  Circle(1.0),
  Square(4)
]

# Print out each shape's area
for shape in shapes
  log(shape.area())

```

## people.cross

```

class Person(name, age)
  fun birthday => age += 1
  fun about => "$name is $age years old"

class Employee(name, age, title) as Person
  fun about => "$name is a $age year-old $title"

class SalariedEmployee(salary) as Employee
  fun about => "$base.about making $salary/yr"

class HourlyEmployee(rate) as Employee
  fun about => "$base.about making $rate/hr"

class Manager(dept, reports) as SalariedEmployee
  fun about => "$base.about as the manager of $dept"
  fun getReports: reports.each(r => r.about())

bob = Person('Bob', 42)
bill = Employee('Bill', 51, 'Janitor')

```

```
brad = HourlyEmployee('Brad', 25, 'Mechanic', 19.50)
ben = SalariedEmployee('Ben', 29, 'Executive', 150000)
bart = Manager('Bart', 49, 'Manager', 300000, 'IT', [bill, brad, ben])

log(bart.getReports())
```