Puma Programming Language Specification

Anthony Burchfield

Release

**Version 1.00**

**June 2024**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 03/15/2023 | 0.01 | Begin documenting of the language. | Anthony Burchfield |
| 6/26/2024 | 1.00 | First release | Anthony Burchfield |
|  |  |  |  |

Copyright and Intellectual Property Notice

This document, “Puma Programming Language Specification”, the file Puma Programming Language Specification.docx, the Puma Programming Language itself and the Puma Programming Language name are the intellectual property of and is Copyrighted © 2024 by Darryl Anthony Burchfield.

Creator and original architect of the Puma Programming Language: Darryl Anthony Burchfield.

Table of Contents

[Abstract 5](#_Toc170502230)

[Paradigms 5](#_Toc170502231)

[Goals 5](#_Toc170502232)

[Rules 5](#_Toc170502233)

[Features 6](#_Toc170502234)

[Supported 6](#_Toc170502235)

[Not supported 6](#_Toc170502236)

[Glossary of Terms 6](#_Toc170502237)

[Language Syntax 7](#_Toc170502238)

[Reserve Words 7](#_Toc170502239)

[Grammar Notation 8](#_Toc170502240)

[Source Files 8](#_Toc170502241)

[Comment Line 9](#_Toc170502242)

[Use Section 9](#_Toc170502243)

[Type/Trait/Module Section 11](#_Toc170502244)

[EnumsSection 13](#_Toc170502245)

[Properties Section 14](#_Toc170502246)

[Access Modifiers 16](#_Toc170502247)

[Mutability Modifiers 16](#_Toc170502248)

[Constant Modifier 17](#_Toc170502249)

[Initialize/Start Sections 17](#_Toc170502250)

[Finalize Sections 18](#_Toc170502251)

[Functions Section 18](#_Toc170502252)

[Statement Block 19](#_Toc170502253)

[Object Expression 23](#_Toc170502254)

[Function Calls 24](#_Toc170502255)

[Compound Statements 24](#_Toc170502256)

[Branch Statements 24](#_Toc170502257)

[Loop Statements 26](#_Toc170502258)

[Error Handle 27](#_Toc170502259)

[Property, Parameter and Variable Declarations 28](#_Toc170502260)

[Identifier 28](#_Toc170502261)

[Basic Types 28](#_Toc170502262)

[Literals 28](#_Toc170502263)

[Integer 28](#_Toc170502264)

[Real 29](#_Toc170502265)

[Boolean 29](#_Toc170502266)

[Character 29](#_Toc170502267)

[String 30](#_Toc170502268)

[Basic Base Types 31](#_Toc170502269)

[Containers 31](#_Toc170502270)

[Sequence Initializers 31](#_Toc170502271)

[Implicit Casting 31](#_Toc170502272)

[Memory Management 32](#_Toc170502273)

[Display 32](#_Toc170502274)

[Libraries 32](#_Toc170502275)

[Coding Style Convention 33](#_Toc170502276)

[Example Code 33](#_Toc170502277)

[Explanation of Language Design 36](#_Toc170502278)

# Abstract

Puma is a programming language that enables the developer to write code that is organized and maintainable. The design also focuses on readability, reliability and efficiency. It is both a procedural and object-oriented programming language. Memory management is handled by the compiler during the build. Types are organized into different files. Puma’s string type supports the Unicode character set in a fast and efficient way. Reference types are never null. Thread safety is supported.

Puma is more than just the features it supports. Puma is also the features it does not support. Puma was architected to help the developer avoid patterns of writing that reduce maintainability.

# [Paradigms](https://en.wikipedia.org/wiki/Programming_paradigm)

The Puma programming language is a general-purpose [high-level](https://en.wikipedia.org/wiki/High-level_programming_language) [programming language](https://en.wikipedia.org/wiki/Programming_language) supporting multiple [paradigms](https://en.wikipedia.org/wiki/Programming_paradigm).

* Static typing
* Type-safe
* [Lexically scoped](https://en.wikipedia.org/wiki/Lexically_scoped)
* [Imperative](https://en.wikipedia.org/wiki/Imperative_programming)
* [Procedural](https://en.wikipedia.org/wiki/Functional_programming)
* [Object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming)

# Goals

1. High level of organization
2. High level of maintainability
3. High level of readability
4. High level of reliability
5. High level of efficiency

# Rules

1. No ugly syntax.
   1. Puma uses a limited amount of punctuation. The punctuation used must not stand out.
   2. The order and location where keywords are placed must not reduce readability.
2. All exception must be caught.
   1. Either don’t support throwing exceptions, catch all exceptions within a function, or create a new construct that resembles throwing exceptions but safer to use.
3. No null references.
   1. All references are valid.
4. No switching on type.
   1. The TypeOf () function won’t be supported.
5. Defaults must be chosen to increase ease of use and learning. Advance settings and features may be added but not be required to write code.
6. One type-define per file by design.
   1. Enums are the exception.
7. All types can be inherited.
8. All functions in base types must support default behavior.
9. All functions in base types can be overridden.

# Features

## Supported

* Clean simple syntax.
* Organized code.
* Object oriented and procedural programming.
* Single Type and multi-trait inheritance.
* Polymorphism.
* Owner/Borrower memory management.
* Fast and efficient Unicode string type.
* HTML window displays generated by Puma library calls.
* Mutable/Immutable objects.

## Not supported

Puma must restrict or not include features that goes against the rules.

* Branching on type.
  + This reduces organization and maintainability.
  + Puma supports dynamic generics instead.
* Base types with no default behavior.
  + This increases redundant code.
* Static generic syntax.
  + Dynamic generics is more organized and maintainable.
* Sealed types.
  + Inheritance is an important part of the Puma programming language.

# Glossary of Terms

Value type – an object type that is passed between variables, parameters and properties by value.

Object type – an object type that is passed between variables, parameters and properties by reference. The reference addresses the object. This is equivalent to passing the object.

Function – A subroutine that can be called (invoked) by other routines.

Method – A function contained within an object or type. Puma uses the keyword function for both functions and methods.

Parameters – Variables that receive the values or objects pass into the function.

Arguments – Values or objects that gets passed into a function.

Properties – Variables that belong to a type or file. Sometimes referred to as fields.

Object-oriented programming – is a programming paradigm based on the concept of *objects* which can contain data (fields, properties) and procedures (functions, methods).

Procedural programming – is a programming paradigm that involves implementing the behavior of a computer program as procedures (functions, methods) that call (invoke) each other.

EOL – End-of-line marker.

EOF – End-of-file.

# Language Syntax

Statements ends at an end-of-line marker. Statements that need to wrap to the next line will have an escape sequence consisting of a backslash followed by an end-of-line marker. Compound statements consisting of a header followed by a block of statements. The header section ends at an end-of-line marker. The block of statements ends with the ***end*** keyword. If a compound statement has only one statement within the block, the entire compound statement can optionally be placed on one line. Examples of compound statements include; if, elseif, else, while, for in, match when, begin and loop statements. The condition expression for the if, elseif and while follow the keyword and end at the end-of-line marker.

# Reserve Words

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| use | as |  |  |  |
| type | trait | module | is | has |
| value | object | enums | base |  |
| properties | functions | start | initialize | finalize |
| return | yield |  |  |  |
| public | private | Internal | override |  |
| var | constant | readonly | readwrite |  |
| int | i64 | i32 | i16 | i8 |
| uint | u64 | u32 | u16 | u8 |
| float | f64 | f32 |  |  |
| fixed | fx64 | fx32 |  |  |
| char | str | bool |  |  |
| implicit | explicit | operator |  |  |
| get | set | with |  |  |
| if | elseif | else |  |  |
| and | or | not |  |  |
| for | in | while | repeat |  |
| begin | end | break | continue |  |
| match | when |  |  |  |
| multithread | multiprocess |  |  |  |

# Grammar Notation

The lexical and syntactic grammars are presented using grammar productions. Each grammar production defines a nonterminal symbol and the possible expansions of the symbol. In the expansion, the abbreviation *opt* means optional. Camel case words are grammar productions. Lower case words are keywords except the italicized abbreviation *opt*. Comments within a grammar notation begin with a hash character (#) and are not included in the expansion.

# 

# Source Files

Source files are formatted as UTF-8 files with the puma file extension (filename.puma). These files consist of up to seven sections; use, type, enums, properties, initialize, finalize, and functions. The type section has two alternative, trait and module. The initialize section has one alternative, start. Each of these sections are optional except. The sections that are included in a file must be in the order listed. A section ends where the next section begins. The exception is the last section within a file ends at the **end** keywork. The **end** keyword is optional only if there are no sections in the file. A file where all the code is commented out is treated as an empty file.

If a source file does not end in a EOL marker, an EOL marker must be appended to the file.

Grammar Production

SourceFiles: # One or more UTF-8 source files that get built into one executable or library.

* SourceFile EOF SourceFiles
* SourceFile EOF

SourceFile:

* UseSection opt TypeTraitModuleSection opt EnumsSection opt PropertiesSection opt

InitializeStartSection opt FinalizeSection opt FunctionsSection opt end

## Comment Line

Comments can be inserted anywhere in a source file. There are two types of comments; single line comments and multiline comments. Single line comments start with a double forward slash and end at the end-of-line marker. Multiline comments start with a forward slash followed by a period and end with a period followed by a forward slash. Multiline comments are able to wrap to the next line; however, multiline comments can also be within the middle of a line with code before and/or after the comment.

Grammar Production

*CommentLine*:

* // Text EOL
* /. Test ./

EOL:

* CR
* LF
* CR LF
* NL

CR:

* U+000D

LF:

* U+000A

NL:

* U+0085

## Use Section

The optional use section begins with the ***use*** keyword that is on a line by itself. The use section imports the namespace of zero or more files. The statements that follow the use section header contains paths to puma source files, C language files or compiled libraries. The paths for the imported files have forward slash delimiters separating the directory names followed by a file name and file extension separated by a period. The compiler will convert from forward slash delimiters to back slash delimiters when necessary.

Grammar Production

UseSection:

* use EOL UseStatementBlock

UseStatementBlock:

* UseStatement UseStatementBlock
* EOF // empty line

UseStatement:

* UseFilePath EOL

UseFilePath:

* UseFullFilePath as Aliase
* UseFullFilePath

Aliase:

* Identifier

UseFullFilePath:

* DirectoryPath / FileName.FileExtension
* FileName.FileExtension

DirectoryPath:

* DirectoryName / DirectoryPath
* DirectoryName

FileName:

* FileCharacterSequence

FileCharacterSequence:

* FileNameCharacter FileCharacterSequence
* FileNameCharacter

FileNameCharacter:

* UTF-8 character

FileExtension:

* puma
* c
* lib
* a

## Type/Trait/Module Section

The optional type section contains a type, trait or module definition. A type definition file defines an object or value type. A trait definition file defines a feature that is inherited by a type. A module definition file defines a set of procedures and data that are group together.

The type section is a single line that starts with the ***type***, ***trait*** or ***module*** keyword. Thekeyword is followed by the name of the type, trait or module being defined. The name includes zero or more namespace names separated by periods. The Type section declares that the entire file is a definition of a type, trait or module. Only one type, trait or module definition can be included per source file.

For the type definition, the name must be followed by the ***is*** keyword and a base type name. One and only one base type can be inherited. There are two basic base types available; value and object. Other types can be used as the base type instead of the basic base types. Optionally, the base type name can be followed by the ***has*** keyword and one or more trait names separated by commas. The base type and optional traits are inherited by the type being defined.

Trait and module definitions cannot inherit a base type or traits. Also, a trait or module cannot be instantiated.

Value types inherit other value types or the base type Value. Object types inherit other object types or the base type Object. A Value type object is passed between variables, parameters and properties by value. An Object type object is passed between variables, parameters and properties by reference. The reference addresses the object.

Grammar Production

TypeTraitModuleSection:

* type TypeName Inheritance
* trait TraitName EOL
* module ModuleName EOL

Inheritance

* is BaseType has TraitList EOL
* is BaseType EOL

BaseType:

* value
* object
* PredefinedType

TraitList:

* PredefinedTrait, TraitList
* PredefinedTrait

TypeName:

* Identifier.TypeName
* Identifier

TraitName:

* Identifier.TraitName
* Identifier

ModuleName:

* Identifier.ModuleName
* Identifier

Identifier:

* IdentifierCharacterSerquence

IdentifierCharacterSerquence:

* IdentifierFirstCharacter IdentifierCharacterContinued
* IdentifierFirstCharacter

IdentifierCharacterContinued:

* IdentifierCharacter IdentifierCharacterContinued
* IdentifierCharacter
* IdentifierFirstCharacter:
* Any letter from any alphabet supported by Unicode

IdentifierCharacter:

* Any letter from any alphabet supported by Unicode
* Any decimal number supported by Unicode
* \_

## EnumsSection

An enumeration type (Enums) is a value type defined by a set of named constants. Enums are the only type that have their own section and their own definition syntax. This allows for multiple enums to be defined in one file including in the same file as another type definition.

The optional enums section begins with the ***enums*** keyword that is on a line by itself. The enum section consists of zero or more enum definitions. Each definition starts with a name on a line by itself and is followed by member names. The members can be in constant assignment statements or not assigned. If not assigned, the enum members will start at zero and increment from member to member. Enums default to global public access.

Grammar Production

EnumsSection:

* enums EOL EnumDefinitions

EnumDefinitions:

* EnumDefinition EnumDefinitions
* EOL // empty line

EnumDefinition:

* EnumName is Type EOL EnumDeclarationBlock
* EnumName EOL EnumDeclarationBlock

EnumName:

* Identifier

EnumDeclarationBlock:

* EnumsDeclaration EnumDeclarationBlock
* end EOL

EnumsDeclaration:

* EnumMemberName = ConstantExpression EOL
* EnumMemberName EOL

EnumMemberName:

* Identifier

## Properties Section

Properties are variables that are associated with the file or type defined in the same file. The optional properties section begins with the ***properties*** keyword that is on a line by itself. The properties section consists of zero or more assignment statements that declare and initialize properties at compiler time. The assignment statements in this section contain a property name followed by an equal sign followed by a literal or object constructor. Literals can be one of the following types, integer, floating point, fixed point, boolean, character, string, array, tuple, list or dictionary. Literals may be followed by an optional abbreviated type declaration. Optionally, properties may be simply declared by following the property name by the ***is*** keyword, followed by a type. Object constructors contain a defined type name followed by parenthesis. The parenthesis may contain argument literals that will be used by the initialization routine to initialize the object type or left empty. Properties may also be dynamically initialized in the initialize section or start section of the file. The property initializations defaults to all zeros unless assigned in the properties, initialize or start sections. The properties also default to private access.

Grammar Production

PropertiesSection:

* properties DefaultModifiers EOL PropertyDeclarationBlock
* properties EOL PropertyDeclarationBlock

DefaultModifiers:

* DefaultAccessModifier, DefaultMutabilityModifier
* DefaultAccessModifier
* DefaultMutabilityModifier

DefaultAccessModifier:

* CompoundAccessModifier

PropertyDeclarationBlock:

* PropertyDeclaration EOL PropertyDeclarationBlock
* EOL // empty line

PropertyDeclaration:

* VariableName = ConstantExpression PropertyModifiers
* VariableName = ConstantExpression
* VariableName = Type PropertyModifiers
* VariableName = Type

DefaultMutabilityModifier:

* MutabilityModifier

PropertyModifiers:

* CompoundAccessModifier MutabilityModifier
* CompoundAccessModifier
* MutabilityModifier

CompoundAccessModifier:

* internal AccessModifier
* AccessModifier

## Access Modifiers

The private access modifier makes functions and enums accessible only by the functions within the type or file they are defined and any derived type. The public access modifier makes the properties accessible by any function within the application. The internal access modifier makes the properties, functions and enums accessible only from within a library. The default for functions is public and not internal. The default for properties is private and not internal.

Note: It is recommended to use the default access without added the keyword.

Grammar Production

AccessModifier:

* public
* private

## Mutability Modifiers

Mutability modifiers declare the object that a variable references as mutable or immutable by the variable. The default mutability for objects is mutable.

The **readonly** keyword declares the object as immutable by the variable that references it. This means that the readonly variables cannot modify the objects it references. However, the variable can be reassigned to another object. When assigning a readonly object to another variable, the object remains readonly unless the **readwrite** keyword is used.

The **readwrite** keyword changes the mutability of the object from immutable to mutable when assigning to another variable. However, the original variable will remain unable to modify the object.

Note: The **readonly** and **readwrite** keywords are used with object type variables.

Note: The above rule applies to variables and properties.

## Constant Modifier

The **constant** keyword declares a variable to be immutable. This means that the constant variable itself cannot be modified. When assigning a constant variable to another variable, the assigned variable will be mutable unless the **readonly** keyword is used.

The **constant** keyword and the **readonly** keywork together declare the variable and the object it references to both be immutable.

Note: The above rule applies to variables and properties.

Note: The **constant** keyword can be used with object type and value type variables.

Grammar Production

MutabilityModifier:

* readonly
* readwrite
* constant

## Initialize/Start Sections

The initialize and start sections are used to initialize the properties and resources at run-time. The initialize section in each procedural file is executed before the start function. Only one file within an application can have a start section.

The initialize section in type or trait files is optional; however, type and trait files may have multiple initialize section with different parameter list. The initialize section within a type or trait file is executed when an object is created.

The Start section is executed after the initialize sections of the procedural files and contains the startup routine and main routine combined. The Start section can only appear once in an application and only in a procedural (module) file. Therefore, every application must have at least one procedural source file. The initialize section defaults to public access.

Grammar Production

InitializeStartSection:

* initialize ( ParameterList ) EOL StatementBlock
* initialize EOL StatementBlock
* start ( string[] ) EOL StatementBlock
* start EOL StatementBlock

## Finalize Sections

The finalize section is used to release resources. The finalize section within a type or trait file is executed when an object is destroyed.

Grammar *Production*

FinalizeSection:

* finalize EOL StatementBlock

## Functions Section

Functions are subroutines that are executed when called by other routines or by recursive calls to itself. Functions can have a variable number of parameters that receives values and objects from the calling routine. Functions can have a variable number of return values or objects (tuples). Value types are passed by value and object types are passed by reference to the parameter list or from the return of the function. Functions contained within a type definition are known as methods.

The optional functions section starts with the keywork functions on a line by itself. Each function definition starts at the beginning of a different line with the name of the function followed by parenthesis followed by an optional type placed after a colon. The parenthesis may contain zero or more comma delimited parameters. Parameters start with the name of the parameter followed by the is keyword followed by a type. Each function header is followed by a statement block. The statement block ends at the ***end*** keyword. The statement block within a function definition is indented. Functions default to object or file public access.

Grammar Production

FunctionsSection:

* functions DefaultModifiers EOL FunctionDefinitions
* functions EOL FunctionDefinitions

FunctionDefinitions:

* FunctionDefinition FunctionDefinitions
* EOF // empty line

FunctionDefinition:

* FunctionName ( ParameterList opt ) return Type EOL StatementBlock
* FunctionName ( ParameterList opt ) EOL StatementBlock
* AccessModifier FunctionName ( ParameterList opt ) return Type EOL StatementBlock
* AccessModifier FunctionName ( ParameterList opt ) EOL StatementBlock

FunctionName:

* Identifier

ParameterList:

* ParameterDeclaration , ParameterList
* ParameterDeclaration

ParameterDeclaration:

* ParameterName = ConstantExpression ParameterModifiers
* ParameterName = ConstantExpression
* ParameterName Type ParameterModifiers
* ParameterName Type

ParameterModifiers:

* MutabilityModifier

## Statement Block

Grammar *Production*

StatementBlock:

* Statement StatementBlock
* end EOL

Statement:

* AssignmentStatement
* FunctionCall
* IfStatements
* MatchStatement
* BeginStatement
* WhileStatement
* ForStatement
* RepeatStatement

AssignmentStatement:

* AssignmentExpression EOL

AssignmentExpression:

* VariableList AssignmentOperator MultiExpression

VariableList:

* VariableName , VariableList
* \_ , VariableList
* VariableName
* \_

VariableName:

* Identifier

MultiExpression:

* Expression as VariableModifiers , MultiExpression
* Expression , MultiExpression
* Expression as VariableModifiers
* Expression

VariableModifiers:

* MutabilityModifier

Expression:

* ConditionalExpression

ConditionalExpression:

* ( LogicalOrExpression ConditionalOperator ConditionalExpression )
* LogicalOrExpression ConditionalOperator ConditionalExpression
* LogicalOrExpression

LogicalOrExpression:

* ( LogicalAndExpression LogicalOrOperator LogicalOrExpression )
* LogicalAndExpression LogicalOrOperator LogicalOrExpression
* LogicalAndExpression

LogicalAndExpression:

* ( EqualityExpression LogicalAndOperator LogicalAndExpression )
* EqualityExpression LogicalAndOperator LogicalAndExpression
* EqualityExpression

EqualityExpression:

* ( RelationalExpression EqualityOperator RelationalExpression )
* RelationalExpression EqualityOperator RelationalExpression
* RelationalExpression

RelationalExpression:

* ( BitwiseOrExpression RelationalOperator BitwiseOrExpression )
* BitwiseOrExpression RelationalOperator BitwiseOrExpression
* BitwiseOrExpression

BitwiseOrExpression:

* ( BitwiseXorExpression BitwiseOrOperator BitwiseOrExpression ) Type
* ( BitwiseXorExpression BitwiseOrOperator BitwiseOrExpression )
* BitwiseXorExpression BitwiseOrOperator BitwiseOrExpression
* BitwiseXorExpression

BitwiseXorExpression:

* ( BitwiseAndExpression BitwiseXorOperator BitwiseXorExpression ) Type
* ( BitwiseAndExpression BitwiseXorOperator BitwiseXorExpression )
* BitwiseAndExpression BitwiseXorOperator BitwiseXorExpression
* BitwiseAndExpression

BitwiseAndExpression:

* ( ShiftExpression BitwiseAndOperator BitwiseAndExpression ) Type
* ( ShiftExpression BitwiseAndOperator BitwiseAndExpression )
* ShiftExpression BitwiseAndOperator BitwiseAndExpression
* ShiftExpression

ShiftExpression:

* ( AdditiveExpression ShiftOperator ShiftExpression ) Type
* ( AdditiveExpression ShiftOperator ShiftExpression )
* AdditiveExpression ShiftOperator ShiftExpression
* AdditiveExpression

AdditiveExpression:

* ( MultiplicativeExpression AdditiveOperator AdditiveExpression ) Type
* ( MultiplicativeExpression AdditiveOperator AdditiveExpression )
* MultiplicativeExpression AdditiveOperator AdditiveExpression
* MultiplicativeExpression

MultiplicativeExpression:

* ( CastingExpression MultiplicativeOperator MultiplicativeExpression ) Type
* ( CastingExpression MultiplicativeOperator MultiplicativeExpression )
* CastingExpression MultiplicativeOperator MultiplicativeExpression
* UnaryExpression

UnaryExpression:

* UnaryOperator UnaryExpression Type
* ( UnaryOperator UnaryExpression )
* UnaryOperator UnaryExpression
* Literal Type
* Postfix

Postfix:

* MemberAccess PostfixOperator Type
* MemberAccess PostfixOperator
* MemberAccess

MemberAccess:

* PrimaryExpression . MemberAccess Type
* PrimaryExpression . MemberAccess
* PrimaryExpression

PrimaryExpression:

* PrimaryExpression ( ArgumentList ) Type
* PrimaryExpression ( ArgumentList )
* PrimaryExpression [ IndexExpression ] Type
* PrimaryExpression [ IndexExpression ]
* Object

Object

* Identifier Type
* Identifier

IndexExpression:

* Expression # evaluates to an unsigned integer.

ArgumentList:

* MultiExpression

## Object Expression

Type and Trait properties and methods are accessed using a dot notation that precedes the identifier. This dot represents the object’s reference. This is equivalent to a this-reference or self-reference followed by a dot, followed by a property or method identifier.

Expression Precedence Table

|  |  |  |
| --- | --- | --- |
| Grouping | () | Inner to outer |
| Member Access | x.y | Left to right |
| Primary | f(x) a[i] | Left to right |
| Postfix | ++ -- | Only one consecutive postfix |
| Unary | ! - ~ & ++ -- | Right to left. No repeating. |
| Pair, Range | : .. | Only one consecutive pair or range expression |
| Multiplicative | / \* % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> >>> | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Relational | < > <= >= | Only one consecutive relational expression. |
| Equality | == != | Only one consecutive Equality expression. |
| Logical NOT | not | Left to right |
| Logical AND | and | Left to right |
| Logical OR | or | Left to right |
| Conditional | if else | Left to right |
| Multi-expression (tuple) | , | Left to right |
| Assignment, Compound assignment | = /= \*= %= += -= <<= >>= >>>= &= ^= |= | Only one assignment operator per statement. |

## Function Calls

*Grammer Production*

FunctionCall:

* VariableName . FunctionName ( ArgumentList opt ) EOL
* FunctionName ( ArgumentList opt ) EOL

## Compound Statements

Compound statements are statements that begin with a header followed by a statement block. The Header starts with a keywork and can have a conditional expression that follows.

### Branch Statements

The branch statements include the if, elseif, else and match statements.

Grammar Production

IfStatements:

* IfStatement ElseStatements
* IfStatement

ElseStatements:

* ElseIfStatement ElseStatements
* ElseIfStatement
* ElseStatement

IfStatement:

* if BoolianExpression Statement # single statement
* if BoolianExpression EOL StatementBlockForIf

ElseIfStatement:

* elseif BoolianExpression EOL StatementBlockForIf
* elseif BoolianExpression Statement # single statement

ElseStatement:

* else Statement # single statement
* else EOL StatementBlock

BoolianExpression:

* LogicalOrExpression # result is true or false.

StatementBlockForIf:

* Statement StatementBlockForIf # one or more statements
* ElseIfStatement
* ElseStatement
* end EOL

MatchStatement:

* match UnaryExpression EOL WhenStatements

WhenStatements:

* when ConstantExpression EOL StatementBlockForWhen
* end EOL

StatementBlockForWhen:

* WhenStatements # more when statements
* Statement StatementBlockForWhen # one or more statements

BeginStatement:

* begin EOL StatementBlock

### Loop Statements

There are three types of loop statements. The while loop, for loop and repeat loop statements. The while loop will loop until the condition statement equates false. The for loop will loop though the entire container and the repeat loop statement will loop indefinitely unless there is a break within that breaks out of the loop.

Grammar Production

WhileStatement:

* while BoolianExpression Statement
* while BoolianExpression EOL StatementBlock

ForStatement:

* for VariableName in ContainerType Statement
* for VariableName in ContainerType EOL StatementBlock

RepeatStatement:

* repeat Statement
* repeat EOL StatementBlock

BreakStatement:

* break UnsignedInteger
* break

ContinueStatement:

* Continue UnsignedInteger
* Continue

### Error Handle

The error and handle statements give an option to jump down to another section of code when there is an error that cannot be handled gracefully with other statements. They resemble the throw and catch statements of other languages except no exception is throw.

Grammar Production

ErrorStatement:

* error ErrorDescription EOL

ErrorDescription:

* StringLiteral

HandleStatement:

* handle ErrorDescriptionVaraible EOL StatementBlock
* handle EOL StatementBlock

ErrorDescriptionVaraible

* StringVariable

## Property, Parameter and Variable Declarations

There are several built in types including signed and unsigned integers, floating point, fixed point, boolean, characters and strings. All of the built-in types are value types except the string which is an immutable object type which resembles a value type.

Integers can be 8-bit, 16-bit, 32-bit or 64-bit in size. Floats are single precision or double precision IEEE floats. Decimal and fixed-point floats might be supported in later version of the language. Characters are 32-bit Unicode. The size of the Booleans is to be announced. Strings are a flexible type that fit into the smaller of a one-byte, two-byte or four-byte array of characters where all of the characters are the same size within a string. This makes the string efficient in size and speed.

## Identifier

Identifiers begin with an alphabetic character ( a..z, A..Z, U-00C0 .. U-10FFFF, . ) followed by zero or more Alpha-numeric characters and underlines ( a..z, A..Z, U-00C0 .. U-10FFFF, 0..9, \_ ). Leading underscores are not supported.

# Basic Types

The basic types include integers, floating-point, characters, strings, boolean and enum. These predefined types have keywords associated with them. The Enum types are special value types with special features. These special features include auto increment assigning properties. Optionally, the properties can be assigned specific values. Enum properties are constant and cannot be changed in the initialize section.

## Literals

All of the basic types have literals that can be used to declare and assign to variables.

## Integer

There are two integer types supported; signed and unsigned integers. Both integer types are available in four different bit sizes; 8, 16, 32 and 64. Constants can be used to declare and assign these types. The reserve words are listed in the table below.

Table of Basic Integer Types

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Literal |
| int64 or int | 64-bit signed integer | 0 or 0 i64 |
| int32 | 32-bit signed integer | 0 i32 |
| int16 | 16-bit signed integer | 0 i16 |
| int8 | 8-bit signed integer | 0 i8 |
| uint64 or uint | 64-bit signed integer | 0 u, 0FF h, 77 o, 1010 b,  0 u64, 0 h64, 0 o64, 0 b64 |
| uint32 | 32-bit signed integer | 0 u32, 0FF h32, 77 o32, 1010 b32 |
| uint16 | 16-bit signed integer | 0 u16, 0FF h16, 77 o16, 1010 b16 |
| uint8 | 8-bit signed integer | 0 u8, 0FF h8, 77 o8, 1010 b8 |

## Real

There are two real number types supported; floating-point and fixed-point. Both real types are available in two sizes; 32 and 64 bits. Constants can be used to declare and assign these types. The keywords are listed in the table below.

Table of Basic Floating-Point Types

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Literal |
| float64 or float | 64-bit floating-point | 0.0 or 0.0 f64 |
| float32 | 32-bit floating-point | 0.0 f32 |
| fixed64 or fixed | 64-bit fixed-point | 0.0 fx or 0.0 fx64 |
| fixed32 | 32-bit fixed-point | 0.00 fx32 or 0.00 fx32.2 |

## Boolean

The boolean type is supported. The reserve words, true and false, are used to declare and set its value. In the Puma programming language, bools are not integers; they only have two values and cannot be converted to or from integers by casting. The relational, equality and logical expression result in a boolean. Compound statements require the conditional expression to result in a boolean.

Table of Boolean Type

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Literal |
| bool | Boolean value | false, true |

## Character

An individual Unicode character is supported. This character is a 32-bit code point of the Unicode standard and can represent any single Unicode character as well as non-character code points. It meets the UTF-32 standard.

Table of the Character Type

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Literal |
| char | Unicode character | ‘ ’, ‘A’, ‘\u0000’, ‘\u10FFFF’, ‘\u10ffff’ |

## String

Puma strings are Unicode strings optimized for both speed and size. To optimize the Unicode string, Puma strings represent Unicode character strings in one of three forms; a one-byte array, a two-byte array or a four-byte array. The exact size is determined when the string is loaded into the string object.

With Puma strings, each character is of the same size. This size constrain allows for faster processing of the strings. The three sizes allow for a smaller memory size. The internal representation is determined automatically by the string object according to the smallest size that all the characters within the string will fit. Once the string is set, it becomes immutable. If a string needs to be modified by the code, a new string object is created. The string type can be converted to and from UTF8, UTF16, UTF32, ASCII and other less common character set strings through function within the string type.

The string type is passed by reference but acks more like a value type object. This is accomplished by making string types immutable. String type objects are replaced instead of modified; therefore, strings must be created new for each change. Multiple modifications can be optimized within one operation. For example, multiple concatenations can be optimized to produce only one new string.

Puma string Escape Sequence are the same as the C language.

Table of the String Type

|  |  |  |
| --- | --- | --- |
| Keyword | Description | Literal |
| str | String of characters | “”, “ “, “ABC” |

From UTF-8

When converting from UTF-8 Unicode strings, if all the characters are one-byte UTF-8 characters, then it is stored as a one-byte string. If all of the characters are one or two bytes and the two-byte characters have a bit pattern of 110000xx 10xxxxxx, then it is also stored as a one-byte string. If any of the characters are two- or three-byte UTF-8 characters but does not meet the previous condition, then it is stored as a two-byte string. If any of the characters are four-byte characters, then it is stored as a four-byte string.

From UTF-16

When converting from UTF-16 Unicode strings, if all of the characters are less than or equal to 255, then it is stored as a one-byte string. If greater than or equal to 256 and none of the characters are surrogate pairs, then the string is stored as a two-byte string. If any of the characters are surrogate pairs, then the string is stored as a four-byte string.

From UTF-32

When converting from UTF-32 Unicode strings, if the entire string will fit into a one-byte or two-byte string, then the entire string is converted the a one-byte or two-byte string, else it is copied unchanged to a four-byte string.

## Basic Base Types

All types have a basic base type. The basic base types are value, object. Value type variables are assigned by value and contain one or more values. Object type variables are assigned by reference to an object and contain the reference to the object.

## Containers

There are several basic container types. These types include array, tuple, list and dictionary. Each of these basic container types have literals that can define and be assigned to a variable.

Table of Containers

|  |  |
| --- | --- |
| Container type | Literal |
| Array | [1, 2, 3, 4] |
| Tuple | (1, “Name”, true) |
| List | {“One”, “Two”, “Three”} |
| Dictionary | {“One” : 1, “Two” : 2, “Three” : 3} |

## Sequence Initializers

There are literals that define sequences. They can be contained within literals of arrays, tuples, list and dictionaries. These sequences can be used to declare and be assign to variables. Sequences can also be iterated within a for loop statement.

Table of Sequences

|  |  |
| --- | --- |
| Sequential range | [ 1..10 ], ( 1..10 ), { 1..10 } |
| Initialize range | [ 0 \* 10 ], ( 0 \* 10 ), { 0 \* 10 } |

# Implicit Casting

Value types can be Implicitly casted to larger value types as long as the results are the same values. This includes unsigned integers implicitly casted to larger signed integers. Also, integers can also implicitly casted to floating points when the mantissa has a larger number of bits than the integer. Explicit casting is possible between any numerical value type and any other numerical value type.

Implicit casting is also available between a derived type and its base type. Explicit casting between a base type and its derived types are not supported in the Puma programming language.

Table of Implicit and Explicit Casting

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Uint8 | Uint16 | Uint32 | Uint64 | Int8 | Int16 | Int32 | Int64 | Float32 | Float64 | Fixed32 | Fixed64 |
| Uint8 | I | I | I | I | E | I | I | I | I | I | E | E |
| Uint16 | E | I | I | I | E | E | I | I | I | I | E | E |
| Uint32 | E | E | I | I | E | E | E | I | E | I | E | E |
| Uint64 | E | E | E | I | E | E | E | E | E | E | E | E |
| Int8 | E | E | E | E | I | I | I | I | I | I | E | E |
| Int16 | E | E | E | E | E | I | I | I | I | I | E | E |
| Int32 | E | E | E | E | E | E | I | I | E | I | E | E |
| Int64 | E | E | E | E | E | E | E | I | E | E | E | E |
| Float32 | E | E | E | E | E | E | E | E | I | E | E | E |
| Float64 | E | E | E | E | E | E | E | E | E | I | E | E |
| Fixed32 | E | E | E | E | E | E | E | E | E | E | I | I |
| Fixed64 | E | E | E | E | E | E | E | E | E | E | E | I |

# Memory Management

Memory management is accomplished by an Owner/Borrower scheme. Owners are variables in the outer most scope that references a particular object type. Borrowers are variables or parameters in one or more inner scopes from the owner. When an owner goes out of scope, the object is deallocated; also, when an owner is reassigned, the original object is deallocated before the new object is assigned. When a borrower goes out of scope or is reassigned, no deallocation is needed because the object is still being referenced by an outer scope owner.

Multi-owners are two or more outer scope variables that reference the same object. When multi-owners go out of scope, they are compared to see if they still reference the same object. If they reference the same object, it is deallocated. If they reference two difference objects, both objects are deallocated.

# Display

Puma supports generating HTML displays by calling Puma library functions. The software developer doesn’t need to know HTML, just Puma and its libraries. After generating the HTML display, the Puma code will show the display in a thin client. This feature also supports generating web pages.

The methods that update the displays will generate signals that will run methods on the same thread as the displays.

# Libraries

Puma imports libraries that perform common task like reading and writing files, opening and closing ports and more. Common file formats supported are, UTF-8, XML, INI, JSON as well as common databases like SQL, MySQL, NoSQL, MongoDB. Common ports that are supported are, Ethernet, UART, USB.

The Puma compiler is able to generate libraries from Puma code. Prewritten libraries can be imported into a project during compiler time as well as import of dynamically linked libraries during run time.

# Coding Style Convention

There are two coding styles that are supported, camel case and snake case.

For camel case, local variables and parameters are lower camel case (lowerCamelCase). Properties, functions, types and traits names are upper camel case (UpperCamelCase). Constants are upper case with underscores. Leading underscores are not supported.

For snake case, identifiers are lower case or underscore (lower\_snake\_case). Leading underscores are not supported.

Keywords are always lower case.

# Example Code

This is a simple example of how to write Puma code.

// Top of Sound.puma file

trait Sound

functions

Sound() is str

end

end

// Top of Fur.puma file

trait Fur

functions

Fur() is str

end

end

// Top of Pet.puma file

use Sound.puma

use Fur.puma

type Pet is object has Sound, Fur

// Executes before initialize

properties

Name is string

Count = 0 is global public

Size = “”

end

initialize ( name = “Unknown”, size = “Unknown” )

Name = name

Count++

Size = size

end

// Top of Dog.puma file

use Pet.puma

type Dog is Pet

initialize ( name is string )

base( name )

end

functions

Sound() is str

return “bark bark”

end

Fur() is str

return “curly”

end

end

// Top of Cat.puma file

use Pet.puma

type Cat is Pet

initialize ( name is string )

base( name )

end

functions

Sound() is str

return “meow”

end

Fur() is str

return “soft”

end

end

// Top of PetApp.puma file

use Dog.puma

use Cat.puma

// Executes before start

properties

firstPet = Dog( “Rover” )

secondPet = Cat( “Socks” )

end

start // Parameters are optional

writeInfo( .firstPet )

writeInfo( .secondPet )

writeSound( .firstPet )

writeSound( .secondPet )

writeFur( .firstPet )

writeFur( .secondPet )

end

# Explanation of Language Design

Puma has been designed to write organized, maintainable readable and scalable. It has been designed to avoid issues from pour programming styles.