Puma Programming Language Specification

Anthony Burchfield

Version 1.01

February 2025

Revision History

Date	Version	Description	Author
03/15/2023	0.01	Begin documenting the language.	Anthony Burchfield
12/30/2024	1.00	First publication	Anthony Burchfield
02/15/2025	1.01	Corrections to wording. Corrections to formatting. Modified reliable code section. Modified rules 2 and 7. Added and modified reserve words. Added the records section. Modified the use of the constant keyword. Modified the description of the error and handle statements. Removed the as keyword for casting.	Anthony Burchfield

Copyright and Intellectual Property Notice

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

The Puma Programming Language is intended to be an open-source project with a steering committee. Darryl Anthony Burchfield will remain the primary member of the steering committee with veto power for twenty years after the release of the compiler that supports all the features in this document. After the release of the compiler that implements all the features in this document, the steering committee may create and copyright the 2.xx and beyond versions of the Puma Programming Language with new features. However, new features beyond those listed in this document may not violate the rules set in this document without the consent of the creator. The 1.xx versions of the Puma Programming Language and its name will remain the property of the creator but is open to the public to write safe, organized and maintainable software and firmware as well as write compilers and other tools for developing Puma programs. The name Puma Programming Language and it’s shortened form, Puma, is open to the public to write about this programming language, including advertising compilers and other tools that support the Puma Programming Language.

Table of Contents

Abstract

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

The Puma Programming Language 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 by not supporting feature that do not help the developer to write safe, organized, and maintainable code.

Paradigms

The Puma programming language is a general-purpose high-level programming language supporting multiple paradigms.

- Static typing
- Type-safe
- Lexically scoped
- Imperative
- Procedural
- Object-oriented

Goals

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

Explanation of Language Design

The Puma Programming Language has been designed to write safe, organized, maintainable, readable, reliable and efficient code. It has been designed to avoid issues from pour programming practices. All features added shall support writing code that meets the goals above. Any feature that does not support the goals will not be supported.

Features that support writing organized code

- Syntax that supports section.
- Syntax that separates type definitions into separate files.
- Properties and functions that are grouped together to form modules and type definitions.

Features that support maintainable code

- Features above that support organized code.
- Features that support object-oriented code.
- Features that do not need to branching on types.

Features that support readability

- Features above that support organized code.
- Syntax with minimum punctuation.
- Keywords that can be understood by novice.

Features that support reliable code

- Non-nullable references.
- Memory management that prevents dangling references and memory leaks.
- Array containers that perform bound checking.
- When throwing exceptions by keyword, all exceptions must be caught.

Features that support efficient code

- A string type where every character in the current string is the same size but where different strings may have different size characters.
- Features that work the same as equitant feature in the C/C++ languages.

Features that support safe code

- Memory management system that supports handling allocating and deallocating memory automatically at build time.
- References always refer to valid objects.
    - References cannot be null or refer to objects that where deallocated.

Rules

1. No ugly syntax.
    a. Puma uses a limited amount of punctuation. The punctuation used shall not stand out.
    b. The order and location where keywords are placed shall not reduce readability.
2. All exceptions thrown by keyword must be caught.
3. No null references.
    a. All references are valid at the point they are used.
    b. The keyword none may be used to produce an instance of a type with all zero fields and a flag that indicates that the object is intended to represent no object.
4. No branching on type.
    a. Switching on type creates a non-generic and unorganized form of coding.
    b. The TypeOf () function won't be supported.
5. Defaults shall be chosen to increase ease of use. Advance settings and features may be added but not be required to write code.
    a. One exception, properties are private by default.
6. One type-define per file by design.
    a. Enums and Records are the exceptions.
7. All types except the basic types can be inherited.
8. All methods in base types shall support default behavior.
9. All methods in base types can be overridden.
    a. The virtual keyword will not be supported but the override keyword will be supported.
10. All features shall keep the code safe, organized and maintainable.
11. The current reference is used only for the object to assign itself to another object.
    a. The current reference cannot reference properties or function.
    b. This rule supports the no ugly syntax rule.

Features

Supported

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

Not supported

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

- Branching on type.
  - This feature reduces organization and maintainability.
  - Puma supports dynamic generics instead.
- Base types with no default behavior.
  - This feature increases unused code.
  - This feature increases redundant code.
- Static generic syntax.
  - Dynamic generics is organized, maintainable and more generic.
- Sealed types.
  - Inheritance is an important part of the Puma programming language. All types except the basic types can be inherited.

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.

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

Method – A function contained within a type definition. Puma uses the keyword functions for both functions and methods.

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

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

Properties – Variables that belong to a type, trait, module or extension. Also 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.

EOS – End-of-section. A section can end at the beginning of the next section or at the ***end*** keyword that is associated with the sections.

EOF – End-of-file. End-of-file is not a character, but instead, the point where there are not more characters to read from a 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 consist 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. Examples of compound statements include; if, else if, else, while, for in, forall in, match when, repeat and begin statements. The condition expression for the if, else if and while follow the keyword and end at the end-of-line marker.

Reserve Words

Note: Not all reserve words below are supported.

use	as			
type	trait	module	is	has
value	object	base		
enums	records			
properties	functions	start	initialize	finalize
return	yield			
public	private	internal	override	delegate
constant	readonly	readwrite		
int128	int64	int32	int16	int8
uint128	uint64	uint32	uint16	uint8
flt128	flt64	flt32		
fix128	fix64	fix32		
char	str	fstr	vstr	
bool	true	false		
hex	oct	bin		
number	implicit	explicit	operator	
get	set	with		
if	else			
and	or	not		
for	in	while	repeat	forall
begin	end	break	continue	
match	when			
error	catch			
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 eight sections; use, type, enums, records, 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. The sections that are included in a file shall be in the order listed. A section ends where the next section begins except the last section. The last section of 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 shall 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 RecordsSection opt PropertiesSection opt
  InitializeStartCreateSection 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

Sections

A file section begins with a section keyword and ends at the beginning of the next section or at the ***end*** keyword associated with the sections.

Use Section

The optional use section begins with the ***use*** keyword that is on a line by itself. The use section imports types, traits and modules defined in other namespaces.

Grammar Production

UseSection:

- use EOL UseStatementBlock
- use UseStatement

UseStatementBlock:

- UseStatement UseStatementBlock

- EOS # End-of-section

UseStatement:

- UseNameSpace as Alias
- UseNameSpace
- UseFilePath

UseNameSpace:

- NameSpaceName . UseNameSpace
- NameSpaceName EOL

UseFilePath:

- FullFilePath

Alias:

- Identifier

FullFilePath:

- DirectoryPath / FileName.FileExtension
- FileName.FileExtension

DirectoryPath:

- DirectoryName / DirectoryPath
- DirectoryName

FileName:

- FileCharacterSequence

FileCharacterSequence:

- FileNameCharacter FileCharacterSequence

- FileNameCharacter

FileNameCharacter:

- UTF-8 character

FileExtension:

- puma
- c
- h
- lib
- a

Type/Trait/Module/Extend Section

The optional type section contains a type, trait, module or extend definition. A type definition file defines an object or value type. A trait definition file defines a feature that can be inherited by a type. A module definition file defines a set of statice procedures and statice data that are group together in a common name space. An extend definition file extends an existing type definition.

The type section is a single line that starts with the ***type***, ***trait***, ***module*** or ***extend*** keyword. The keyword is followed by the name of the type, trait, module or extension 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, module or extension. Only one type, trait, module or extension definition can be included per source file.

For the type definition, the name shall 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, module and extend definitions cannot inherit base types or traits. Also, a trait, module or extend cannot be instantiated.

The extend definition is useful when needing to add virtual methods to a base type or other features to a type without having access to the existing type's code. The name of the extend is the same as the name of the type it extends.

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

Grammar Production

TypeTraitModuleSection:

- type TypeName Inheritance
- trait TraitName EOL
- module ModuleName EOL
- extend TypeTraitModuleName EOL

TypeName:

- Identifier

TraitName:

- Identifier

ModuleName:

- Identifier

TypeTraitModuleName:

- TypeName
- TraitName
- ModuleName

Inheritance

- is BaseType has TraitList EOL
- is BaseType EOL

BaseType:

- value
- object
- a predefined type

TraitList:

- a predefined trait, TraitList
- a predefined trait

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
- _

Enums Section

An enumeration type (Enums) is a value type defined by a set of named constants. Multiple enums can be defined in one file, including in the same file as a 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 on the code, the enum members will be automatically assigned start at zero and increment from member to member. Enums default to public access.

Grammar Production

EnumsSection:

- enums EOL EnumDefinitions

EnumDefinitions:

- EnumDefinition EnumDefinitions
- EOS # End-of-section

EnumDefinition:

- EnumName is Type EOL EnumDeclarationBlock
- EnumName EOL EnumDeclarationBlock

EnumName:

- Identifier

EnumDeclarationBlock:

- EnumsDeclaration EnumDeclarationBlock
- EOS # End-of-section

EnumsDeclaration:

- EnumMemberName = ConstantExpression EOL
- EnumMemberName EOL

EnumMemberName:

- Identifier

Records Section

A record type is a value type defined with a set of members that form a record. Multiple records can be defined in one file, including in the same file as a type definition.

The optional records section begins with the ***records*** keyword that is on a line by itself. The records section consists of zero or more record definitions. Each definition starts with a name on a line by itself and is followed by member names. Records default to public access.

Table of Record Literal

Record	(1, “Name”, true)

Grammar Production

RecordsSection:

- Records EOL RecordDefinitions

RecordDefinitions:

- RecordDefinition RecordDefinitions
- EOS # End-of-section

RecordDefinition:

- RecordName EOL RecordDeclarationBlock

RecordName:

- Identifier

RecordDeclarationBlock:

- RecordDeclaration RecordDeclarationBlock
- EOS # End-of-section

RecordDeclaration:

- RecordMemberName EOL

RecordMemberName:

- Identifier

Properties Section

Properties are variables that are associated with the module 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/or 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, record, list or dictionary. Literals may be followed by an optional abbreviated type declaration. 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 initialization defaults to all zeros unless assigned in the properties, initialize or start sections. All properties that are not initialized in the initialize or start sections get initialized to all zero bytes. 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
- EOS # End-of-section

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, enums and records 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, enums, and records 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 an object that a variable references to be immutable by the variable. However, the variable itself can be reassigned to another object. When assigning a readonly object to another variable, the object remains readonly to the newly assigned variable unless the ***readwrite*** keyword is used.

The ***readwrite*** keyword changes the object 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. If the variable is an object type, the object is also declared as immutable. When assigning a constant value type variable to another variable, the assigned variable will be mutable unless the ***constant*** keyword is used. When assigning a constant object type variable to another variable, the assigned variable will receive a mutable deep copy of the object unless the constant keyword or readonly keyword is used.

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 module files and contains the startup routine. The Start section shall appear once and only once in an application and only within a module file. Therefore, every application shall have at least one module source file. The initialize section defaults to public access.

Grammar *Production*

InitializeStartSection:

- initialize (ParameterList) opt EOL StatementBlock
- start (string[]) opt EOL StatementBlock
- start (string[]) opt Statement

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 (records). 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 often referred to 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 ***return*** keyword and type. The parenthesis may contain zero or more comma delimited parameters. Parameters start with the name of the parameter followed by a type. Optionally, the parameters may have default values which are assigned by the equal sign. 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 public access.

Optionally, delegate declarations may be defined within the functions section. Delegate declarations define a reference to a function or method. Each delegate definition starts at the beginning of a line with the name of the delegate followed by parenthesis followed by the ***delegate*** keyword. The parenthesis may contain zero or more comma delimited parameters. Parameters start with the name of the parameter followed by a type. Delegate declarations do not have a statement block. The delegate definition ends at the end-of-line. Delegates default to public access.

Grammar Production

FunctionsSection:

- functions DefaultModifiers EOL FunctionDefinitions
- functions EOL FunctionDefinitions

FunctionDefinitions:

- FunctionDefinition FunctionDefinitions
- DelegateDefinition FunctionDefinitions
- EOS # End-of-section

FunctionDefinition:

- FunctionName (ParameterList opt) Type EOL StatementBlock
- FunctionName (ParameterList opt) EOL StatementBlock
- AccessModifier FunctionName (ParameterList opt) 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

DelegateDefinition:

- DelegateName (DelegateParameterList opt) EOL

DelegateName:

- Identifier

DelegateParameterList:

- DelegateParameterDeclaration , DelegateParameterList
- DelegateParameterDeclaration

DelegateParameterDeclaration:

- ParameterName Type ParameterModifiers
- ParameterName Type

ParameterName:

- Identifier

ParameterModifiers:

- MutabilityModifier

Block Statement

Grammar Production

StatementBlock:

- Statement StatementBlock
- end EOL

Statement:

- AssignmentStatement
- FunctionCall
- IfStatements
- MatchStatement
- BeginStatement
- WhileStatement
- ForStatement
- ForAllStatement
- RepeatStatement

AssignmentStatement:

- AssignmentExpression EOL

AssignmentExpression:

- VariableList AssignmentOperator MultiExpression

VariableList:

- VariableName , VariableList
- _ , VariableList
- VariableName
- _

VariableName:

- Identifier

MultiExpression:

- Expression VariableModifiers , MultiExpression
- Expression , MultiExpression
- Expression 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:

- (UnaryExpression MultiplicativeOperator MultiplicativeExpression) Type
- (UnaryExpression MultiplicativeOperator MultiplicativeExpression)
- UnaryExpression 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

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 (record)	,	Left to right
Assignment, Compound assignment	= /= *= %= += -= <<= >>= &= ^= |=	Only one assignment operator per statement.

Function Calls

Grammar 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, else if, else and match statements.

Grammar Production

IfStatements:

- IfStatement ElseStatements

- IfStatement

ElseStatements:

- ElseIfStatement ElseStatements
- ElseIfStatement
- ElseStatement

IfStatement:

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

ElseIfStatement:

- else if BoolianExpression EOL StatementBlockForIf
- else if BoolianExpression Statement # single statement

ElseStatement:

- else EOL StatementBlock
- else Statement # single statement

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 # single statement
- while BoolianExpression EOL StatementBlock

ForStatement:

- for VariableName in Container Statement # single statement
- for VariableName in Container EOL StatementBlock

ForAllStatement:

- forall VariableName in Container Statement # single statement
- forall VariableName in Container EOL StatementBlock

RepeatStatement:

- repeat Statement # single statement
- repeat EOL StatementBlock

BreakStatement:

- break UnsignedInteger
- break

ContinueStatement:

- Continue UnsignedInteger
- Continue

Error Catch

The error and catch 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 are the same as throw and catch statements in other languages. However, all exeptions in Puma functions and methods must be caught. Therefore, for every error statement, there must be a handle statement to catch it. Functions and methods that call functions and methods that throw exceptions must catch the exceptions.

Grammar Production

ErrorStatement:

- error ErrorDescription EOL

ErrorDescription:

- StringLiteral

CatchStatement:

- catch ErrorDescriptionVaraible EOL StatementBlock
- catch 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 is used like a value type.

Integers can be 8-bit, 16-bit, 32-bit or 64-bit in size. Floating points are single precision (32-bit) or double precision (64-bit) IEEE floats. Fixed-points are real numbers implemented as an integer with a fixed decimal point per variable or property. Characters are 32-bit Unicode. The size of the Booleans is 8-bits. Strings are a flexible type that fit into the smaller of a one-byte, two-byte or four-byte arrays 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, , fixed-point, characters, strings, boolean, enums and records. 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	64-bit signed integer	0, or 0 int64
int32	32-bit signed integer	0 int32
int16	16-bit signed integer	0 int16
int8	8-bit signed integer	0 int8
uint64	64-bit signed integer	0 uint64, 0FF hex64, 77 oct64, 1010 bin64
uint32	32-bit signed integer	0 uint32, 0FF hex32, 77 oct32, 1010 bin32
uint16	16-bit signed integer	0 uint16, 0FF hex16, 77 oct16, 1010 bin16
uint8	8-bit signed integer	0 uint8, 0FF hex8, 77 oct8, 1010 bin8

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
flt64	64-bit floating-point	0.0, 0.0 flt64 or 0 flt64
flt32	32-bit floating-point	0.0 flt32 or 0 flt32
fix64	64-bit fixed-point	0.00 fix64 or 0 fix64.2
fix32	32-bit fixed-point	0.00 fix32 or 0 fix32.2

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. Characters are 32-bit code points of the Unicode standard and can represent any single Unicode character as well as non-character code points.

Note: Invalid code points are not checked or enforced. As of the writing of this document, Unicode code points are defined in the range of U+0000 to U+10FFFF. This is only 21 bits out of a 32-bit char. Values outside of this range are undefined and should be avoided in Puma characters and strings. Also, the range U+D800 to U+DFFF is defined in Unicode as surrogate pairs used in UTF-16 files and should also be avoided in Puma characters and strings.

Table of the Character Type

Keyword	Description	Literal
char	Unicode character	‘ ’, ‘A’, 0000 char, 10FFFF char, 10ffff char

String

Puma strings are Unicode strings optimized for both speed and size. To optimize the Unicode string, Puma strings are in one of three forms: a one-byte array (ASCII plus Latin-1), a two-byte array (BMP) or a four-byte array (UTF-32). The exact size is determined when the string is assigned to the string variable. The three string formats are derived types. Polymorphism is used to make them ack like the same object type.

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 ASCII, UTF-8, UTF-16 and UTF-32 through functions within the string type. The default for UTF-8 is no byte order marker as per the recommendations of the standard committee. The defaults for UTF-16 and UTF-32 are big-endian with byte order markers.

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 shall 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.

Note: Invalid code points are not checked or enforced.

Table of the String Type

Keyword	Description	Literal
str	String of characters	“”, “ “, “ABC”
fstr	Format string	“Name: {name}”
vstr	Verbatim string	“c:\directory\file.ext”

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 to a one-byte or two-byte string, else it is copied unchanged to a four-byte string.

Fstr

An fstr is a string containing formatting code. The fstr contain variable, property or function names surrounded by braces. The braces and the names are replaced by the compiler at run time by calling ToString() for each name. Formatting parameters can optionally follow the names.

Example

age = 21

height = 1.8

WriteLine(“Age: {age}\n Height: {height:f0.2}” fstr)

VStr

A vstr is a string that it written verbatim. No formatting will be performed and no escape sequences will be converted.

Example

WriteLine("c:\directory\file.ext" vstring);

Basic Base Types

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

Base Value Type Methods

Size() – size of the type in bytes.

ToString() – returns a printable representation of the value type.

FromString() – takes a printable representation of the value type.

ToBytes() – returns the bytes from the value as a big-endian byte array.

ToBytes(LITTLE_ENDIAN) – returns the bytes from the value as a little-endian byte array.

ToObject() – creates a reference type object from the value type (boxing).

Base Object Type Methods

Size() – size of the type in bytes.

ToString() – returns a printable representation of the value type.

FromString() – takes a printable representation of the value type.

ToBytes() – returns the bytes from the value as a big-endian byte array.

ToBytes(LITTLE_ENDIAN) – returns the bytes from the value as a little-endian byte array.

Copy() – makes a shallow copy of the object.

CopyAll() – makes a deep copy of the object.

ToValue() – creates a value type object from the reference type object (unboxing).Reference() – returns a reference to the object.

Numerical Type

Numerical types like integers, floats and fixed inherit the type number which inherits value. This enables the numerical types to work in dynamic generic functions.

Containers

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

Table of Container Literals

Container type	Literal
Array	[1, 2, 3, 4]
List	{“One”, “Two”, “Three”}
Dictionary	{“One” : 1, “Two” : 2, “Three” : 3}

Array Containers

The array container contains a variable length array. The defaults indexing in Puma is one based indexing. The indexing for array containers can be changed to zero based indexing when needed.

List Containers

The list container contains a linked list of objects. This linking can be single or double linked.

Dictionary Containers

The dictionary container contains a linked list of key value pairs. This linking can be single or double linked.

Sequence Initializers

There are literals that define sequences. They can be contained within literals of arrays, records, 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/Explicit Casting

Value types can be Implicitly casted to larger value types as long as the result is the same value. This includes unsigned integers implicitly casted to larger signed integers. Also, integers can implicitly casted

to floating points when the mantissa has the same or 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 (down-casting). Explicit casting between a base type and its derived types (up-casting) is not supported in the Puma programming language.

Explicit casting is done with ***type*** keywords that follow the expression being typed. Not all cast are valid.

Table of Implicit and Explicit Casting of Numerical Type

	uint8	uint16	uint32	uint64	int8	int16	int32	int64	flt32	flt64	fix32	fix64
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
flt32	E	E	E	E	E	E	E	E	I	E	E	E
flt64	E	E	E	E	E	E	E	E	E	I	E	E
fix32	E	E	E	E	E	E	E	E	E	E	I	I
fix64	E	E	E	E	E	E	E	E	E	E	E	I

Memory Management

Memory management is accomplished by an Owner/Borrower model. Owners are variables or properties in the outer most scope that references a particular object. Borrowers are variables or parameters in inner scope 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.

Co-owners are two or more outer scope variables that reside in the same scope and reference the same object. When co-owners go out of scope, they are compared to each other to see if they still reference the same object. If they reference the same object, the object is deallocated only once. If they reference two or more difference objects, all 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's display library. After generating the HTML display, the Puma code will show the display in a thin client. This feature also supports generating web pages.

The functions that update the displays generate signals that run functions 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 that can be supported include, UTF-8, XML, INI, JSON as well as common databases like MySQL, NoSQL, MongoDB. Common ports that can be supported include, Ethernet, UART, USB.

The Puma compiler is able to generate libraries from Puma code. Prewritten libraries can be imported into a project during build time.

Coding Conventions

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

For camel case, local variables and parameters are lower camel case (lowerCamelCase). Enums, 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 and underscores (lower_snake_case). Constants are upper case with underscores. Leading underscores are not supported but a trailing underscore is 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() string
      return “No sound”
   end
end

// Top of Fur.puma file
trait Fur
functions
   Fur() string
      return “No fur”
   end
end

// Top of Pet.puma file
use
   Sound.puma
   Fur.puma
type Pet is object has Sound, Fur
// Executes before initialize
properties
   Name = string
   Count = 0 public type
   Size = “”
```

```
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 string )
   base( name )

functions
   Sound() string
      return “bark bark”
   end

   Fur()  string
      return “curly”
   end
end

// Top of Cat.puma file
use Pet.puma
type Cat is Pet

initialize ( name string )
```

```
   base( name )

functions
   Sound() string
      return “meow”
   end

   Fur()  string
      return “soft”
   end
end

// Top of PetApp.puma file
use
   Dog.puma
   Cat.puma

// Executes before start
properties
   FirstPet = Dog( “Rover” )
   SecondPet = Cat( “Socks” )

start  // Parameters are optional
   writeInfo( FirstPet )
   writeInfo( SecondPet )

   writeSound( FirstPet )
   writeSound( SecondPet )
```

```
    writeFur( FirstPet )
    writeFur( SecondPet )
end
```