Puma Programming Language Specification

Anthony Burchfield

Version 1.08

Nov 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	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
02/28/2025	1.02	Replaces the string type with a UTF-8 string type. Replaces the char type with a UTF-8 char type. Modified the wording of other types.	Anthony Burchfield
03/10/2025	1.03	Removed the extend section. Modified wording in the string section.	Anthony Burchfield
04/10/2025	1.04	Added nullable types and statement blocks where they are safe to use. Modified rule 3. Modified the feature section. Added boxing/unboxing	Anthony Burchfield
4/27/2025	1.05	Replaced the Reference() method with the ***self*** keyword. Updated wording in the description of the Puma sections.	Anthony Burchfield
8/17/2025	1.06	Added more example code. Change the keyword ***have*** to ***has***. Update the Coding Conventions section. Updated the copyright notice. Updated the Explanation of Language Design.	Anthony Burchfield
10/17/2025	1.07	Updated the Coding Conventions section. Added the pack keyword for packing records. Updated the has statement. Added a has trait statement.	Anthony Burchfield
11/24/2025	1.08	Updated the Memory Management section.	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. 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 non-nullable by default. 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 and maintainable code. It is also intended to be readable, reliable and efficient. 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 may be understood by novice.

Features that support reliable code

- Non-nullable references as the default.
- Statement blocks where nullable types can be safely dereferenced.
- 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

- 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.

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. Reference types are non-nullable by default.
    a. Nullable references are optional.
    b. Nullable reference types can only be dereferenced from within statements that checks for null.
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 may be inherited.
8. All methods in base types shall support default behavior.
9. All methods in base types may 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. A function in the base object type returns the reference to the current object and is used only for the object to assign itself to another object.
    a. This reference shall be used to reference properties or function with in the current object.
    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.
- Non-nullable and Nullable references.

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 may 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 may 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 or module. Also referred to as fields.

Object-oriented programming – is a programming paradigm based on the concept of *objects* which may 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 may 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 end 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 ends 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	number	optional	
enums	records	pack			
properties	functions	start	initialize	finalize	
return	yield				
public	private	internal	override	delegate	
constant	readonly	readwrite			
int128	int64	int32	int16	int8	int
uint128	uint64	uint32	uint16	uint8	uint
flt128	flt64	flt32	flt		
fix128	fix64	fix32	fix		
char	str	fstr	vstr		
bool	true	false			
hex	oct	bin			
implicit	explicit	operator			
get	set	with	self		
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 may 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 may 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

NEL:

- 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. The use section imports the namespaces of types, traits and modules defined in other files.

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 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 may 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.

The type section is a single line that starts with the ***type***, ***trait*** or ***module*** keyword. The keyword 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 may 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 may be inherited. There are two basic base types available; value and object. Other types may be used as the base type instead of the basic base types. Optionally, the base type name may 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 may not inherit base types or traits. Also, a trait or module may not be instantiated.

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

TypeName:

- Identifier

TraitName:

- Identifier

ModuleName:

- Identifier

TypeTraitModuleName:

- TypeName
- TraitName
- ModuleName

Inheritance

- is BaseType has TraitList EOL
- is BaseType EOL

BaseType:

- value
- object
- predefined type

TraitList:

- predefined trait, TraitList
- 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 may be defined in one file, including in the same file as a type definition.

The optional enums section begins with the ***enums*** keyword. 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 may 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 may be defined in one file, including in the same file as a type definition.

The optional records section begins with the ***records*** keyword. The records section consists of zero or more record definitions. Each definition starts with a record name at the beginning of a line and is followed by member names in following lines. Records default to public access. If the keyword **packed** followed by a number follows the record name, then the recorded will be packed. Gaps in the record must be smaller than the number that follows the key word pack.

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 pack UnsignedInteger 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. 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 may be one of the following types, integer, floating point, fixed-point, boolean, character, string, array, record, list or map. Literals may be followed by an optional 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 opt MutabilityModifier opt OptionalModifier opt

Access Modifiers

There are three access modifiers: private, public and internal. 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 internal access modifier may be paired with the private or public access modifiers or used by itself. 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

CompoundAccessModifier:

- internal AccessModifier
- AccessModifier
- internal

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 may 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 may be used with object type and value type variables.

Grammar Production

MutabilityModifier:

- readonly
- readwrite
- constant

Optional Modifier

References default to non-nullable. The ***optional*** keyword declares the reference to be a nullable type. Optional nullable types can only be dereferenced from within the has statement. Dereferencing an

optional reference outside of a has statement will generate a compiler error. Value types that are declared optional are boxed.

Grammar Production

OptionalModifier:

- optional

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 may 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 may have a variable number of parameters that receives values and objects from the calling routine. Functions may 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 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
- HasStatement

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 may 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 statement within the statement block that breaks out of the loop. The ***break*** keyword may contain an integer literal that defines how many loops the break will break out of. The ***continue*** keyword is used to continue at the top of the loop without finishing the current pass through the loop. The ***continue*** keyword may contain an integer literal that specifies which loop that will be continued.

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 Statements

The error and catch statements give the option to jump down to another line of code when there is an error that cannot be handled gracefully with other methods. This is the same as throw and catch statements in other languages. However, all errors thrown in Puma functions and methods must be caught. Therefore, for every error statement, there must be a catch statement to catch it. Functions and methods that call functions and methods that throw errors must catch the errors. The catch statement must catch all errors and exceptions. After catching an error or exception, the type of error or exception may be determined using a parameter in the catch statement. The catch parameter is of type string.

Grammar Production

ErrorStatement:

- error ErrorDescription EOL

ErrorDescription:

- StringLiteral

CatchStatement:

- catch ErrorDescriptionVaraible EOL StatementBlock
- catch EOL StatementBlock

ErrorDescriptionVaraible

- StringVariable

Has Statement

The has statement is used with nullable reference types. The has statement header checks for a non-null reference. If the reference is not null, the statement or statement block is executed. If the reference is null, the statement or statement block is skipped. If a variable or property is not a reference type, a compiler error is generated. Multiple reference can be checked in the same has statement.

Grammar Production

HasStatement:

- has VariableName Statement
- has VariableName EOL StatementBlock end
- has VariableName EOL StatementBlock HasStatement

Has trait Statement

The has statement can also check for traits. If the trait is included in the object, the statement or statement block is executed. If the reference does not have the trait, the statement or statement block is skipped. The reference is also check for null in has trait statement. Multiple traits can be checked in the same has trait statement.

Grammar Production

HasTraitStatement:

- has trait VariableName Statement
- has trait VariableName EOL StatementBlock end
- has trait VariableName EOL StatementBlock HasTraitStatement

Property, Parameter and Variable Declarations

The properties and variables are declared by assigning to a literal, a ***type*** keyword or a defined type. Parameters and function returns are declared with a following ***type*** keyword or defined type.

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, 0..9, _, U+00C0 .. U+10FFFF). Leading underscores are not supported in camelCase but are supported in _snake_case.

Basic Types

The basic types include integer, floating-point, fixed-point, character, string, boolean, enum and record. 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 may be assigned specific values. Enum properties are constant and may not be changed

in the initialize section. All of the basic types are value types except the string which is an immutable object type.

Literals

All of the basic types have literals that may 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 may be used to declare and assign these types. The keywords integers 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
Int	64-bit signed integer	0, or 0 int
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
uint	64-bit signed integer	0 uint, 0FF hex, 77 oct, 1010 bin

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 may 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
flt	64-bit floating-point	0.0, 0.0 flt or 0 flt

fix64	64-bit fixed-point	0.00 fix64 or 0 fix64.2
fix32	32-bit fixed-point	0.00 fix32 or 0 fix32.2
fix	64-bit fixed-point	0.00 fix or 0 fix.2

Boolean

The boolean type is a discrete type with two reserve words, true and false, to declare and set its value. The relational, equality and logical expression result in a boolean. The if statement and while loop require the conditional expression to result in a boolean. The keyword for a boolean is bool.

Table of Boolean Type

Keyword	Description	Literal
bool	Boolean value	false, true

Character

The character type is a single UTF-8 Encoded Unicode character (code point). It is stored in a four-byte array. The keyword for a character is char.

Note: A Puma character hold a single Unicode code point. Some code points may be combined to form a single character. Multiple code points that form one character should be stored in a string instead of a character.

Note: Characters are compared to characters in strings one byte at a time; however, characters are assigned to another character 32-bits at one time.

Table of the Character Type

Keyword	Description	Literal
char	Single character	‘’, ‘ ‘, ‘A’

String

The string type is a UTF-8 encoded Unicode string. String objects consist of a union of two records. One for short strings and the other for long strings. The short string record contains a byte with the most significant bit set to zero, 3 reserved bits and a 4-bit right justified count followed by 15 bytes for the internal string. The long string record contains a byte with the most significant bit set to one, 3 reserved bytes, a 32-bit count and a 64-bit pointer. For 32-bit processor, the 64-bit pointer is replaced with 4 reserve bytes followed by a 32-bit pointer. The long string type is immutable; therefore, modifications are performed by creating a new string. Multiple modifications may be optimized by combining into one operation. The keywords for the different types of strings are listed in the table below.

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

Note: Invalid code points are not checked or enforced by the string type itself; however, the valid ranges from U+0000 to U+D7FF, U+E000 to U+10FFFF will be checked and enforced by the standard libraries. Code points outside of these ranges will be replaced by the libraries with the replacement character U+FFFD code point. Overlong characters will also be checked and replaced. However; the noncharacters in the range of U+FDD0 to U+FDEF as well as the code points at the end of each plane with the least significant byte of FE hex and FF hex will not be treated as invalid nor replaced. Puma libraries must follow these recommendations of the Unicode Consortium.

Table of the String Type

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

Str

Standard strings are strings that may contain escape sequences. The escape sequences are converted to Unicode control characters.

Fstr

Format strings are strings containing formatting code as well as escape sequences. The fstr contain variable, property or function names surrounded by braces. The braces and the names are replaced at run time by calling ToString() for each named identifier or function return type. Formatting parameters may optionally follow the identifier names. Formatting parameters supported are the same as the C programming language.

Note: Format strings format themselves at run time and do not need a function to perform the formatting.

Example

```
age = 21

height = 1.8

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

Vstr

Verbatim strings are strings that are written verbatim. No formatting will be performed and no escape sequences will be converted.

Example

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

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 object in bytes.

SizeReference() – size of the reference to the object.

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

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

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

ToBytes(LITTLE_ENDIAN) – returns the bytes from the object 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).

Self – reference to itself, the object. Self cannot be dereferenced. It is only used for assignments and function arguments.

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 record, array, list and map. Each of these basic container types have literals that may define and be assigned to a variable.

Table of Container Literals

Container type	Literal
Record	(int, str, bool), (1, “Name”, true)
Array	[int], [int * 10 * 10], [1, 2, 3, 4]
List	{str}, {“One”, “Two”, “Three”}
Map	{str : int}, {“One” : 1, “Two” : 2, “Three” : 3}

Array Containers

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

List Containers

The list container contains a variable length array. The internal array grows as needed to hold the list.

Map Containers

The map container contains a variable length array of key value pairs. The internal array grows as needed to hold the list.

Sequence Initializers

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

Table of Sequences

Sequential range	[1..10], (1..10), { 1..10 }, { 1:””..10}
Initialize range	[0 * 10], (0 * 10), { 0 * 10 }, { 0:””*10}

Implicit/Explicit Casting

Value types may 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 may 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

Boxing/Unboxing

Value types may be boxed by casting to object types using the keyword ***object***. Object types may be unboxed by casting to value types using the keyword ***value***. Boxing and Unboxing may be done anywhere where casting is valid.

Memory Management

Memory management uses the Ownership Model, where owners are outermost variables or properties referencing an object, and borrowers are inner scope variables or parameters. An object is deallocated when its owner leaves scope or is reassigned; if a borrower leaves scope or is reassigned, the object remains since the owner still references it.

Co-owners are multiple outer scope variables in the same scope referencing the same object. When they go out of scope, they are checked: if still referencing the same object, it's deallocated once; if not, all referenced objects are deallocated. If a co-owner is reassigned, ownership transfers to the other.

Lifetime anchors are backing code used as the owner when a local or global variable cannot be used safely as the owner of the object.

Dynamic ownership is determined at runtime using a backing code flag to mark a variable or property as the owner or co-owner of an object. Dynamic ownership should only be used if the other forms of ownership cannot be used safely.

The compiler should evaluate all memory management options and select the most suitable method for each code section where an object appears. It may use different techniques across sections, as no one approach fits every scenario.

Display

Puma supports generating HTML displays by calling Puma library functions. The software developer doesn't need to know HTML, just the Puma's display library. After generating the HTML display, the Puma code will show the display in a thin client.

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

The Puma compiler is able to generate libraries from Puma code. Prewritten libraries may be imported into a project during build time. Dynamically linked libraries are also supported.

Note: The default for UTF-8 in text files is no byte order marker as per the recommendations of the standard committee. The defaults for UTF-16 and UTF-32 in text files are big-endian with byte order markers. Little-endian files and no byte order markers are optional.

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, public properties, public functions, types and traits names are upper camel case (UpperCamelCase). Private properties and private functions are lower camel case. Constants are upper case without underscores (MAXLENGTH). Leading underscores are not supported.

For snake case, identifiers are lower case with underscores (lower_snake_case). Constants are upper case with underscores (MAX_LENGTH). Leading underscores are supported for private properties and functions but not for public properties and functions.

Keywords are always lower case.

Example Code

Example 1

This is a simple example of a type definition.

```
// This code defines a simple user profile type in the Puma programming language.

// use directives
use System.Console

// Class declaration
type UserProfile is object

// Enum declarations
enums
  StatusSetting
    Active,
    Inactive,
    Pending
  end
```

```
// Struct declarations
records
   UserRecord
      Name string
      Age int
   end

properties
   Status = StatusSetting
   User = UserRecord

// Constructor
initialize (status StatusSetting, user UserRecord)
   Status = status
   User = user

// Destructor
finalize
   // Cleanup code here

// Method declarations
functions
   // Method to display user information
  DisplayUserInfo ()
      WriteLine("Name: {User.Name}, Age: {User.Age}, Status: {status}" fstr)
  end

   // Method to update status
```

```
   UpdateStatus (status StatusSetting)
      Status = status
   end
end
// end if file
```

Example 2

This is a simple example of how to write dynamic generic code in Puma programming language.

```
// Top of Sound.puma file
trait Sound
functions
   Sound() str
      return “No sound”
   end
end
// end of file

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

// Top of Pet.puma file
use
```

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

initialize ( name = “Unknown”, size = “Unknown” )
   Name = name
   Count++
   Size = size
End
// end of file

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

initialize ( name str )
   base( name )

functions
   Sound() str
      return “bark bark”
   end

   Fur()  str
```

```
        return “curly”
    end
end
// end of file

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

initialize ( name str )
    base( name )

functions
    Sound() str
        return “meow”
    end

    Fur()  str
        return “soft”
    end
end
// end of file

// 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
// end of file
```