Petram Language Specification Summary

Core Principles

- Explicitness and verbosity as features
- Strong typing with constraint-based refinements
- Functional programming influences with imperative capabilities
- Focus on data-oriented design

Basic Structure

- Indentation-significant for block scoping
- Lines terminated by newlines (no semicolons)
- Comments use `--` for single line and `{- -}` for multi-line

Variables and Assignments

- Variables prefixed with `$`
- Type inference: `$variable := value`
- Explicit typing: `$variable: Type = value`

Functions and Methods

- Defined: `func #[name :: param1: Type, param2: OtherType, ..., $paramN: TypeN]#: ReturnType ->`
- Single-expression functions use `=>`
- Calls: `#[function_name(arg1: value1, arg2: value2)]#`
- Argument names must always be provided.
- Methods similar to functions but use `method` keyword

Structs and Traits

- Structs: `struct #[Name]# ->`
    - Structs have a constructor with the special `new #[arg1: Type1, $arg2: Type2, ..., $argN: TypeN]#: Self` signature

- To instantiate a struct:

```
struct #[Square]# ->
  field side_length: Float

  new #[side_length: Float]#: Self ->
    @side_length = side_length

-- elsewhere in the code
$square := #[Square::new :: side_length: 3.4]#
```

- Traits: `trait #[Name]# ->`
    - Traits are similar to Protocols in Objective C or interfaces in other OO programming languages.
- Fields: `field name: Type`
- Constrained fields: `constrain struct_field_name: Type where #[condition]# message: "Error message"`
    - If you introduce one or more constraints to your struct, then the return type of the `new #[..]#` constructor changes from `Self` to `Result<Self, String>`, and you must pattern match on it. The string will be the error message you've defined in that particular constraint.
- Trait implementation and other struct inheritance: `struct #[Rectangle < Shape, Printable]# ->`

Control Structures

- We support the standard if/else if/else pattern Note that `if` is an expression and must be enclosed in `#[]#` as it returns a value.
    - The `if else` and final `else` are optional

```
#[
  if #[somecond]# ->
    -- statements, expressions
  -- optionally
  else if #[someothercond]# ->
      -- ...
  -- optionally
  else ->
    -- ...
]#
```

- If you don't want to return anything from the `if` expression, you can discard it with the special _ pattern.

```
_ := #[
      if #[somecond]# ->
          -- statements, expressions
      {- optionally
      else ->
          -- ... -}
    ]#
```

- Loops: `foreach $item in $collection -> ...`

```
-- inferred as List<Int>
$collection := {|1, 2, 3|}

-- $item is inferred as Int
foreach $item in $collection ->
  #[println :: message: "Item: {$item}"]#

-- Prints "Item: 1"
-- Prints "Item: 2"
-- Prints "Item: 3"
```

Pattern Matching

Pattern matching is an expression and therefore must be enclosed in `#[]#` as it returns a value.

```
$somevar := #[
    match $something_else ->
        Pattern1 -> result1
        Pattern2 -> result2
        _ -> default_result
    ]#
```

Generics

- `struct #[List<T>]# ->`

Error Handling

- `Result<T, E>` for operations that can fail
  - `Result::Ok()` will contain the `T`. `Result::Error()` will contain the `E`
- `Option<T>` for when a value might or might not be present. No nulls or nils.
  - Like in rust, there's `Option::Some(value)` and `Option::None`.
- No traditional try/catch syntax, use `#[match]#`

Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`
- Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical: `and`, `or`, `not`
- Pipe: `|>` for method chaining
  - When piping to other functions, the previous variable's name may be omitted:

```
$numbers := {|1, 2, 3|} -- List<Int>

$times_two_squared_divided_by_three := #[
  $numbers.map(f: func #[num: Int]: Int => num * 2)
  |> #[.map(f: func #[num: Int]: Int => num * num)]
  |> #[.map(f: func #[num: Int]: Float => num / 3)]
]#

-- result: {|1.3333333333333333, 5.333333333333333, 12|}
-- $times_two_squared_divided_by_three is inferred as List<Float>
```

String Interpolation

- `"Value: {$variable}"`
- Inside structs: `"Value: {@field_name}"`

Lists and Collections

- Lists are enclosed in `{| |}
- `$numbers := {|1, 2, 3, 4, 5|} -- inferred as List<Int>`

Unique Features

- Constraint-based field validation in structs

- Explicit expression syntax with `#[ ]#`
- Strong emphasis on compile-time checks and constraints

Note: Petram prioritizes explicitness, verbosity, and robustness. It's designed as a general-purpose, low-level language capable of targeting multiple architectures, with a focus on data-oriented design and compile-time safety.