

Related: [Computer Science](#), [Intuition Engineering](#), [Programming](#)

Thing-Action-Context Language, TACL- A General Purpose Language

Specification Version: v1.0 (2023-11-13T05:22)

Taglines:

~~"Hope you like Brackets"~~

~~"Hold on to your Brackets"~~

~~"Bracket Order Matters"~~

~~Everything is an Action~~ Everything is a THING!

"It's All Brackets, All The Way Down!"

Corey Naas, November 2023

<https://coreynaas.com/>

coreynaas@outlook.com

Copyright © 2023 Corey Naas. All rights reserved.

Thing-Action-Context Language (TACL) is a language that centers around the concept of "Things" which encapsulate data, actions, and control structures. TACL emphasizes readability and simplicity, using unique syntax that mirrors natural language while maintaining computational precision.

It is a formulaic way to explain how to do anything.

This is the functional description for Thing-Action-Context Language (TACL), a general purpose language written by Corey Naas on 08 November 2023.

Hello World

In this program, we define a `Main` action that takes no inputs. This action simply prints "Hello World!" to the standard output and then returns 0.

Tacl ▾

```
1  <Main>[DefineAction]{
2      <Input> is {}
3      <Default><Error>"An Error has occurred."
4      <Body>{
5          [Print]{"Hello World!"}
6          <Return>{0}
      }
}
```

Table of Contents

1. [Hello World](#)
2. [Table of Contents](#)
3. [Introduction](#)
4. [Syntax Quick Reference Table](#)
5. [TAC](#)
 1. [Things](#)
 2. [Contexts](#)
 3. [Actions](#)
6. [Showing Things Off](#)
7. [Orderings Things with Contexts](#)
 1. [Concise and Plain Forms](#)
8. [Expanding Things with Boxes and Piles](#)
9. [Controlling the Flow of Things](#)
 1. [If-Then-Else](#)

2. [Loops](#)
 1. [While Loop](#)
 2. [For Loop](#)
3. [Attempt](#)
4. [Switch and Case](#)
10. [Appendix: Standard Library of Actions](#)
 1. [System/Shell/IO Actions](#)
 2. [Control Flow Actions](#)
 3. [Mathematical and Logical Operation Actions](#)
11. [Appendix: Example Code](#)
 1. [Hello World](#)
 2. [Making a Sandwich](#)
 3. [FizzBuzz](#)
 4. [Factorial](#)
 5. [Fibonacci Sequence](#)
 6. [Reverse a String](#)
12. [Appendix: Turing-Completeness Review](#)
13. [Appendix: Syntax Completion Surveys](#)
 1. [Bubble Sort](#)
 1. [Prompt for Reviewer](#)
 2. [Response from Reviewer](#)
 2. [Binary Search Tree](#)
 1. [Prompt for Reviewer](#)
 2. [Response from Reviewer](#)
 3. [Conway's Game of Life](#)
 1. [Prompt for Reviewer](#)
 2. [Response from Reviewer](#)
14. [Appendix: "Syntactic Sugar"](#)
15. [Appendix: Paradigms](#)

Introduction

BEGIN AI-GENERATED CONTENT

Thing-Action-Context Language is designed to be a simple, versatile, and intuitive way to describe "doing stuff with things". It is a broad yet basic syntactic language that transcends the conventional boundaries of the physical and the abstract, the concrete and the conceptual.

In contrast to similar-looking languages such as LISP or TCL, our language does not rely on symbolic expressions or prefix notation for function calls. Instead, it uses a unique and intuitive syntax to represent actions and their parameters. Our language also deviates from pseudocode by introducing specific syntax for control flow statements and error handling, as well as clearly defined contexts for data transportation, assignment, and execution.

Our language is a significant departure from plain English too. While it is designed to be readable in plain English, it also introduces computational syntax and language conventions for clarity and precision. This makes our language more expressive and powerful, without sacrificing readability or understandability.

A salient feature of our language is its use of *Things* as the fundamental units of the language. *Things* can be objects, abstract ideas, numbers, colors, sounds, feelings, tastes, or even actions. Each *Thing* is distinguished with a unique identifier or label, and can represent various components such as inputs, outputs, and actions.

We introduce the concept of "contexts" to provide extra-contextual information about how to interpret and execute the usage of a *Thing*. Our language uses different types of brackets to denote four

different contexts: action context `[]`, transport context `{}`, assignment context `<>`, and index context `()`.

Our language also introduces the concept of "Boxes" and "Piles" to represent collections of *Things*. A *Box* is a loose collection of *Things*, while a *Pile* is an ordered collection of *Things* with associated indexes.

Control flow in our language is represented using actions such as:

- `[Condition]{condition, [ActionIfTrue]{parameters}, [ActionIfFalse]{parameters}}` for if-then-else statements,
- `[While]{{[ConditionAction]{parameters}, [ActionSequence]{parameters}}}` for while loops,
- and `[For]{{[InitializationAction]{parameters}, [ConditionAction]{parameters}, [UpdateAction]{parameters}, [ActionSequence]{parameters}}}` for for loops.

Error handling in our language is implemented through the simplified `[Attempt]` action which attempts an action and handles exceptions in one go. Our language also supports switch/case statements using the `[Switch]` action followed by a Box or Pile of cases.

This language specification serves as a comprehensive guide to our language, providing detailed explanations of its unique features, syntax, and conventions. We hope it will serve as a useful tool for understanding and using our language to its fullest potential.

Regards,
Cora

END AI-GENERATED CONTENT

Syntax Quick Reference Table

Concept	Syntax
Thing	identifier
Thing (Transport)	{identifier}
Thing (Assignment)	<identifier>
Thing (Action)	[identifier]
Action Definition	<Action>[DefineAction]{ <Input>{param1, param2, ...}, <Default><error>"Error message", <Body>{<operations>}}
Action Invocation	<Result>[Action]{param1, param2, ...}
Concise Assignment	<Assignment>[Action]
Action with Transport	[Action] with {param1, param2, ...}
Concise Action-Transport	[Action]{param1, param2, ...}
Box	<Box>{item1, item2, ...}
Pile (Ordered Box)	<Pile>{(1)item1, (2)item2, ...}
Access Pile at Index	[Pile](Index)
If-Then-Else	[Condition] with {condition, [ActionIfTrue]{params}, [ActionIfFalse]{params}}
While Loop	[While]{{[ConditionAction]{params}, [ActionSequence]{params}}}
For Loop	[For]{{[InitializationAction]{params}, [ConditionAction]{params}, [UpdateAction]{params}, [ActionSequence]{params}}}
Attempt	[Attempt]{{[ActionToAttempt]{params}, [ErrorHandler]{params}}}
Switch-Case	[Switch]{value, {(1)[Case]{{(1)case1Value, (2)[Action1]{parameters}}, (2)[Case]{{(1)case2Value, (2)[Action2]{parameters}}, ...}}}

TAC

Things

The syntax for a *Thing* is flexible. It is enclosed in curly braces `{}` when it represents being transported (i.e., read from), such as the inputs to an action, enclosed in angle brackets `<>` when it represents being assigned (i.e., written to), and enclosed in square brackets `[]` when it represents actions (i.e., executed). When referring to a *Thing* within the language, we simply use its identifier.

In the context of our language, a *Thing* can represent various components:

- It can be an input to an action, or assigned to another *Thing*.
- It can be an action, indicating the operations or transformations performed.
- It can be an output from an action, or assigned the value of another *Thing*.

Contexts

In our language, we introduce the concept of "contexts", which are represented by different types of brackets. This concept is rooted in the philosophy of making the language readable in plain English, while also providing extra-contextual information about how to interpret and execute the usage of a *Thing*.

Here are the contexts we've defined:

1. **Action Context** `[]`: This context is all about actions. When you see a *Thing* within square brackets, you know that it represents an operation to be executed.
2. **Transport Context** `{}`: This context transports *Things*, as parameters, to an action. A *Thing* within curly braces is a piece of information or data being passed to the action defined in the square brackets. Multiple *Things* are separated by periods or new lines and indentation.
3. **Assignment Context** `<>`: This context changes the value or content, if any, of a *Thing*. The input can be a primitive, an action, or a transport representing multiple *Things*.
4. **Index Context** `()`: This context contains the index of a *Thing* in a *Pile*.

```
1 # Valid Assignment Context usages
2 <myThing> is "the pen in my right pocket"
3 <myAge> is 27
4 <myStuff> is {myThing, myAge}
5 <allMyStuff>{myStuff, <myName>"Corey"}
6 <allMyStuffOrdered>{(1)myStuff, (2)<myName>"Corey"}
```

Actions

A function is a process or operation that takes certain information (input) and produces a result (output). To simplify our language and make it more intuitive, we replace "Function" with "Action". An "Action" can be seen as a higher-level "Thing" that performs operations and manipulates other "Things".

Here's an example of how you could write a simple action, using the system `[DefineAction]` action:

```
1 <AddNumbers>[DefineAction]{
2     <Input>{num1, num2},
3     <Default><error>"An error has occurred.",
4     <Body>{
5         <result>[Add]{num1, num2}
6         <Return>{result}
7     }
8 }
```

The `[DefineAction]` action takes three parameters in a transport: an Input, a Default output, and a Body.

1. The `<Input>` is a *Thing* that contains the parameters of the action.
2. The `<Default>` is a *Thing* that contains the *Thing* to return if the action doesn't otherwise return. In this example, the string is also being assigned to a *Thing* named "error".
3. The `<Body>` is a *Thing* that encapsulates the operations of the action. This approach retains the transport and assignment aspects of our language, while also simplifying the syntax.

The `<Return>` is a fourth *Thing*, that returns from the action. It is set to `<Default>` when the action is first executed.

We allow eliding the commas in a transport list if the transport list is fully separated by lines.

Showing Things Off

This is an example that demonstrates most of the major features of TACL.

```
TACL v
1  <Main>[DefineAction]{
2      <Input> is {}
3      <Default> is <error> is "An Error has occurred, please panic."
4      <Body> is {
5          # myFavouriteNumber is 7
6          <myFavouriteNumber> is 7
7
8          # Print with myFavouriteNumber
9          [Print] with {myFavouriteNumber}
10
11         # Print with CustomAdd with myFavouriteNumber and 4
12         [Print] with {[CustomAdd] with {myFavouriteNumber, 4}}
13
14         # myFavouriteNumber is 8
15         <myFavouriteNumber>8
16
17         # Print with myFavouriteNumber
18         [Print]{myFavouriteNumber}
19
20         # myNewNumber is myTestAction with myFavouriteNumber.
21         <myNewNumber> is [myTestAction] with {myFavouriteNumber}
22
23         # myNewNumber is myTestAction with Add with myFavouriteNumber and 42.
24         <myNewNumber>[myTestAction]{[Add]{myFavouriteNumber, 42}}
25
26         # Print with myNew Number
27         [Print]{myNewNumber}
28
29         # myNewNextNumber is MyTestAction with myNewNumber
30         <myNextNewNumber> is [MyTestAction] with {myNewNumber}
31
32         # Print with myNextNewNumber
33         [Print] with {myNextNewNumber}
34
35         [Condition]{
36             [GreaterThan]{10,2},
37             [Print]"10 is greater than 2!",
38             [Print]"10 is less than 2!"
39         }
40
41         [Attempt]{[Print]{[Divide]{1,0}}, [Print]"You can't divide by zero!"}
42
43         [Attempt]{
44             [Print]{
45                 [Divide]{2,0}
46             },
47             [Print]"You \*still\* can't divide by zero!!!"
48         }
49
50         <Return>0
51     }
52 }
```

```

}

<CustomAdd>[DefineAction]{
  <Input>{num1, num2}
  <Default><Error>"An Error has occurred, please panic."
  <Body>{
    <temp>[Add]{num1, num2}
    <Return>{[Add]{temp, 1}}
  }
}

<MyTestAction>[DefineAction]{
  <Input>{inputNum}
  <Default>"An Error has occurred, please panic."
  <Body>{
    <internalNumber>7
    <newNumber>[CustomAdd]{inputNum, internalNumber}
    <Return>{newNumber}
  }
}

```

Orderings Things with Contexts

In TACL, the arrangement of assignments, actions, and transport contexts is flexible, yet follows specific rules. This flexibility allows you to construct different types of expressions based on your needs. Here are the valid orders for these components:

1. An Assignment can be any sort of anything that can follow the keyword **is**: `<Assignment> is 5`, or `<Assignment>5` for short. Enclose strings in double quotation marks: `<Assignment> is "my shoe"`, or `<Assignment>"my shoe"`
2. An assignment can be an action: `<Assignment> is [Action]` or `<Assignment>[Action]` for short.
3. An action can take a transport context using the keyword **with**: `[Action] with {Transport}` or concisely `[Action]{Transport}`.
4. An action, if a Pile, can take an index context, which returns that Thing in the Pile: `[myPile] with (3)` returns the third Thing in myPile, or `[myPile](3)` for short.
5. An assignment can be an action with a transport context: `<Assignment> is [Action] with {Transport}` or more concisely `<Assignment>[Action]{Transport}`.
6. A transport can be a comma-separated list of *Things*, actions, or assignments. Indexes, to induce order, can be included using the index context.
7. In Development:
 1. A `(Piece) of [Thing]` ... using the keyword **of**... `(piece)[Thing]`
 2. ... using the keyword **in**...

Concise and Plain Forms

TACL offers two forms for writing expressions: the Plain form and the Concise form. The Plain form uses keywords like 'is', 'with', and 'at' to structure the expression, making it more readable. The Concise form, on the other hand, omits these keywords, resulting in a more compact expression. Here's how the two forms compare:

1. In Plain form, an assignment is an action: `<Assignment> is [Action]`. In Concise form, this becomes `<Assignment>[Action]`.
2. In Plain form, an action takes a transport context: `[Action] with {Transport}`. In Concise form, this becomes `[Action]{Transport}`.
3. In Plain form, an assignment can be an action with a transport context: `<Assignment> is [Action] with {Transport}`. In Concise form, this becomes `<Assignment>[Action]{Transport}`.
4. In Plain form, an assignment can be a transport context: `<Assignment> is {Transport}`. In Concise form, this becomes `<Assignment>{Transport}`.

5. In Plain form, an assignment can be a Pile with *Things* in given indexes using the keyword **at**:
`<Assignment>` is `{"Item 1" at 1, "Item 2" at 2}`. In Concise form, this becomes `<Assignment>{(1)"Item 1", (2)"Item 2"}`.
6. In Plain form, a Pile, in an action context, takes an index context: `[Pile] with (Index)`. In Concise form, this becomes `[Pile](Index)`. For compatibility, either **with** or **at** are accepted in Plain form.
7. In Plain form, an assignment can be a Pile, in an action context, with an index context: `<Assignment>` is `[Pile] with (Index)`. In Concise form, this becomes `<Assignment>[Pile]{Index}`. For compatibility, either **with** or **at** are accepted in Plain form.

Expanding Things with Boxes and Piles

In our language, we use the concept of "Boxes" to represent a collection of *Things*. This is similar to the idea of a "list" or an "array" in many programming languages, but with a key distinction.

A *Box* is a *Thing* that contains other *Things*. The specific *Things* that a *Box* contains are not ordered, sorted, or otherwise structured.

Tact 

```
1 #Box - Plain Form
2 <Thing1> is "Whatever is in my left front pocket right now."
3 <Thing2> is "the memory of watching the sunset in Nassau on a cool fall evening"
4 <myThing> is {Thing1}
5 <myThingBox> is {myThing, Thing2, Thing3}

#Box - Concise Form
<Thing1>"Whatever is in my left front pocket right now."
<myThing>{Thing1}
<myThingBox>{myThing, Thing2, Thing3}
```

We adapt the concept for an ordered *Box*, or *Pile*. Here, we use the **at** keyword or the index context `()` to associate the *Things* with their indexes.

Tact 

```
1 #Pile - Plain Form
2 <myThingPile> is {myThing at 1, [Add]{Thing2, 1} at 2, "This is a thing too!" at 3}
3
4 #Pile - Concise Form
5 <myThingPile>{(1)myThing, (2)[Add]{Thing2, 1}, (3)"This is a thing too!"}
```

Things within a pile can be accessed by their index using the index context `()` or the `[AccessAt]` action:

Tact 

```
1 <myFirstThing>"my will to live"
2
3 <MyFirstPile>{(1)myFirstThing, (2)[Add]{1, 1}, (3)"This is a thing too!"}
4
5 <mySpecificThing>[myFirstPile](1)
6 [Print]{mySpecificThing} # would print "my will to live" to the standard output
7
8 <myThing>[AccessAt]{myThingPile, 3}
9 [Print]{myThing} # would print "This is a thing too!" to the standard output
10
11 <myIndex>1
12 <myThing>[AccessAt]{myThingPile, [Add]{myIndex, 1}}
13 [Print]{myThing} # would print "2" to the standard output
14
```

We allow eliding the commas in a Box or Pile if the contents is fully separated by lines. For example:

```

1  <myThingBox>{
    myThing
    Thing2
    Thing3
}

<myThingPile> is {
    myThing at 1
    Thing2 at 2
    Thing3 at 3
}

<myThingPile>{
    (1)myThing
    (2)[Add]{2, 1}
    (3)"This is a thing too!"
}

```

Controlling the Flow of Things

If-Then-Else

1. **If**: We use the action context `[]` to denote as `[Condition]`.
2. **Then**: The actions to be performed when `condition` is true is represented as a sequence of Things or Actions. We denote this as `[ActionIfTrue]{parameters}`.
3. **Else**: The actions to be performed when `condition` is false can also be represented as a sequence of Things or Action. We denote this as `[ActionIfFalse]{parameters}`.

```

1  [Condition] with {condition, [ActionIfTrue]{parameters}, [ActionIfFalse]{parameters}}

```

In this structure, `[Condition]` is an action that checks the `condition` and executes either the `[ActionIfTrue]{parameters}` sequence of Things if the condition is true, or the `[ActionIfFalse]{parameters}` sequence of Things if the condition is false.

Loops

In our language, we redefine traditional loops using our own unique action and transport contexts. Here's how we approach this:

While Loop

The while loop is represented by an action, `[While]`. The `[While]` action takes two parameters: an action that returns a boolean (the loop condition), and an action (or sequence of Actions) to execute while the condition is true.

```

1  [While]{{[ConditionAction]{parameters}, [ActionSequence]{parameters}}

```

For Loop

The for loop is a bit more complex, as it involves initialization, condition checking, and incrementation/decrementation. We represent this with an action, `[For]`. The `[For]` action takes three parameters: an action that initializes the loop variable, an action that checks the loop condition, and an action that updates the loop variable at the end of each iteration. A fourth parameter is the Action (or sequence of Actions) to execute in each iteration.

```

1  [For]{{[InitializationAction]{parameters}, [ConditionAction]{parameters}, [UpdateAction]{parameters}, [ActionSequence]{parameters}}

```


Attempt

Given our language structure, we upgraded our try-catch mechanism into a single action. This is done by introducing a new action `[Attempt]` which simultaneously attempts an action and handles any exceptions in one go. The `[Attempt]` action can take two parameters: the action to be attempted and the action to be executed in case of an error.

The syntax looks like this:

Tact ▾

```
1  [Attempt]{[ActionToAttempt]{parameters}, [ErrorHandler]{parameters}}
```

For example, if we have an action `CalculateDistance` that takes two parameters `start_point` and `end_point`, and an error handling action `HandleError`, the `[Attempt]` action would look like this:

Tact ▾

```
1  [Attempt]{[CalculateDistance]{start_point, end_point}, [HandleError]{parameters}}
```

In this case, `CalculateDistance` will be attempted. If it executes without an exception, then the program proceeds normally. But if an exception is thrown, the `HandleError` action is called.

Switch and Case

To implement switch/case statements in our language, we use the action `[Switch]` followed by a Thing that we want to evaluate. We then define a Box of `[Cases]`, each represented as a Pile with the case value and the corresponding action.

```
[Switch]{value, {(1)[Case]{(1)case1Value, (2)[Action1]{parameters}}, (2)[Case]{(1)case2Value, (2)[Action2]{parameters}}, ...}}
```

The structure of the context is as follows:

- `{value, {(1)[Case]{(1)case1Value, (2)[Action1]{parameters}}, (2)[Case]{(1)case2Value, (2)[Action2]{parameters}}, ...}}`

Breaking it down:

- `value`: This is the *Thing* being evaluated. The *Thing* could be a numerical value, a string, or any other type of data that can be compared.
- `[(1)[Case]{(1)case1Value, (2)[Action1]{parameters}}]`: This represents a single case. Each case is represented as a Pile within a Box. The case itself is a Pile with two elements: the case value (`case1Value`) and the Action to execute if the case value matches the `value`.
- `(1)case1Value`: This is the value for the case. If `value` matches `case1Value`, then `[Action1]{parameters}` is executed.
- `(2)[Action1]{parameters}`: This is the Action that is executed if `value` matches `case1Value`. The Action takes parameters, which are represented within the transport context `{parameters}`.

The entire `[Switch]` action context contains multiple cases, each defined in the same way. The cases are tested in order, and when a match is found, the corresponding Action is executed.

Here's an example of a simple switch-case statement that tests a numerical value:

Tact ▾

```
1  <Main>[DefineAction]{
2    <Input> is {}
3    <Default><Error>"An Error has occurred."
4    <Body>{
5      <value>2
6      [Switch]{
7        value,
8        {
          (1)[Case]{(1)1, (2)[Print]{"One"}},
          (2)[Case]{(1)2, (2)[Print]{"Two"}},
          (3)[Case]{(1)3, (2)[Print]{"Three"}}
```

```

    }
    }
    <Return>{0}
  }
}

```

In this example, the `value` is 2. The `[Switch]` action context tests `value` against the case values 1, 2, and 3. When it finds that `value` matches the second case (2), it executes the corresponding Action, which is `[Print]{"Two"}`.

Appendix: Standard Library of Actions

System/Shell/IO Actions

- `[Assign]{thing1, thing2}`
 - Assigns `thing1` the contents of `thing2`. This is the same as using the assignment context `< >`.
- `[AccessAt]` with `{Pile, Index}`
 - Returns the *Thing* in the given Pile at the given Index. This is the same as using `[Pile](Index)`.
- `[DefineAction]` with `{Input, Default, Body}`
 - Creates an Action using the given Input(s), Default returned *Thing*, and Body. Returns a *Thing*, which could very well be no*Thing*... null.
- `[Print]` with `{thing1, thing2, ..., thingN}`
 - Prints the contents of `thing1`, `thing2`, ..., `thingN` to the standard output.

Control Flow Actions

- `[Condition]{condition, [ActionIfTrue]{parameters}, [ActionIfFalse]{parameters}}`
 - If-Then-Else.
- `[Switch]{value, {[Case]{{(1) case1Value, (1) [Action1]{parameters}}, [Case]{{(2) case2Value, (2) [Action2]{parameters}}, ...}}`
 - Switch Statement.
- `[Case]{{(1) case1Value, (1) [Action1]{parameters}}`
 - Case Statement. Used in `[Switch]`.
- `[Attempt]{{[ActionToAttempt]{parameters}, [ErrorHandler]{parameters}}`
 - Exception handling.
- `[For]{{[InitializationAction]{parameters}, [ConditionAction]{parameters}, [UpdateAction]{parameters}, [ActionSequence]{parameters}}`
 - For Loop.
- `[While]{{[ConditionAction]{parameters}, [ActionSequence]{parameters}}`
 - While Loop.

Mathematical and Logical Operation Actions

- `[Add]{thing1, thing2, ..., thingN}`
 - Adds `thing1` and `thing2`, and `thing3` ..., and `thingN` and returns the Sum.
- `[Subtract]{thing1, thing2, ..., thingN}`
 - Subtracts `thing1` from `thing2`, and from `thing3`, ..., from `thingN`, and returns the Subtrahend.
- `[Multiply]{thing1, thing2, ..., thingN}`
 - Multiplies `thing1` and `thing2`, and `thing3`, ..., and `thingN` and returns the Product.
- `[Divide]{thing1, thing2, ..., thingN}`
 - Divides `thing1` by `thing2`, by `thing3`, ..., by `thingN` and returns the Dividend.
- `[Increment]{thing, increment}`
 - Increment `thing` by `increment`.
- `[Decrement]{thing, decrement}`
 - Decrement `thing` by `decrement`.
- `[IsEven]{thing}`
 - Returns true if `thing` is even, false if `thing` is odd.

- `[And]{thing1, thing2, ..., thingN}`
 - Returns true if thing1, thing2, ..., thingN are all true, returns false if any are false.
- `[Or]{thing1, thing2, ..., thingN}`
 - Returns true if any of thing1, thing2, ..., thingN are true, returns false if all are false.
- `[Not]{thing}`
 - Returns true if thing is false, returns false if thing is true.
- `[GreaterThan]{thing1, thing2, ..., thingN}`
 - Returns true if thing1 is greater than thing2, and if thing2 is greater than thing3, ..., and if thingN-1 is greater than thingN.
- `[LessThan]{thing1, thing2, ..., thingN}`
 - Returns true if thing1 is less than thing2, and if thing2 is less than thing3, ..., and if thingN-1 is less than thingN.
- `[Equals]{thing1, thing2, ..., thingN}`
 - Returns true if thing1 is equal to thing2, and if thing2 is equal to thing3, ..., and if thingN-1 is equal to thingN.
- `[Unequals]{thing1, thing2, ..., thingN}`
 - Returns true if thing1 is unequal to thing2, and if unthing2 is equal to thing3, ..., and if thingN-1 is unequal to thingN.

Appendix: Example Code

BEGIN AI-GENERATED CONTENT

Hello World

In this program, we define a `Main` action that takes no inputs. This action simply prints "Hello World!" to the standard output and then returns 0.

```

1  <Main>[DefineAction]{
2      <Input> is {}
3      <Default><Error>"An Error has occurred."
4      <Body>{
5          [Print]{"Hello World!"}
6          <Return>{0}
7      }
8  }
```

Making a Sandwich

In this TACL script, the process of making a peanut butter and jelly sandwich is broken down into several actions:

- `MakeSandwich`: The main action that outlines the steps to make the sandwich.
- `GetBreadSlice`: An action to get a slice of bread.
- `Spread`: An action to spread either peanut butter or jelly on a slice of bread.
- `Combine`: Combines two bread slices into a sandwich.
- `Serve`: Presents the final sandwich.

Each action is defined using the `[DefineAction]` action which includes the inputs required, a default error message, and the body containing the steps for that action. The `<Return>` is used to specify what each action should return upon completion.

The script assumes the existence of actions `TakeSlice`, `ApplySpread`, `PlaceOnTop`, and `Present` within the environment, which perform the literal operations of taking a bread slice, spreading an ingredient, combining two slices, and serving the sandwich, respectively.

```
1  <MakeSandwich>[DefineAction]{
2      <Input>{}
3      <Default><error>"Unable to make sandwich due to an error."
4      <Body>{
5          # Get two slices of bread
6          <breadSlice1> is [GetBreadSlice]{}
7          <breadSlice2> is [GetBreadSlice]{}
8
9          # Spread peanut butter on one slice
10         [Spread]{peanutButter, breadSlice1}
11
12         # Spread jelly on the other slice
13         [Spread]{jelly, breadSlice2}
14
15         # Put the two slices together
16         <sandwich>[Combine]{breadSlice1, breadSlice2}
17
18         # Sandwich is ready to be served
19         [Serve]{sandwich}
20
21         <Return>0
22     }
23 }
24
25 <GetBreadSlice>[DefineAction]{
26     <Input>{}
27     <Default><error>"No bread available."
28     <Body>{
29         <slice>[TakeSlice]{bread}
30         <Return>{slice}
31     }
32 }
33
34 <Spread>[DefineAction]{
35     <Input>{spread, breadSlice}
36     <Default><error>"Spread or bread not available."
37     <Body>{
38         [ApplySpread]{spread, breadSlice}
39         <Return>{breadSlice}
40     }
41 }
42
43 <Combine>[DefineAction]{
44     <Input>{slice1, slice2}
45     <Default><error>"Cannot combine slices."
46     <Body>{
47         <combined>[PlaceOnTop]{slice1, slice2}
48         <Return>{combined}
49     }
50 }
51
52 <Serve>[DefineAction]{
53     <Input>{item}
54     <Default><error>"Nothing to serve."
55     <Body>{
56         [Present]{item}
57     }
58 }
```

```

        <Return>0
    }
}

# Start making the sandwich
[MakeSandwich]{}

```

FizzBuzz

The FizzBuzz program prints the numbers from 1 to 100, but for multiples of 3 prints "Fizz" instead of the number, and for multiples of 5 prints "Buzz". For numbers which are multiples of both 3 and 5 prints "FizzBuzz".

The `IsMultiple` action checks if a number is a multiple of a base number. If it is, it returns "Is a multiple.", otherwise it returns "Not a multiple.".

The `FizzBuzz` action checks if a number is a multiple of 3, 5, or 15, and then prints "Fizz", "Buzz", or "FizzBuzz" accordingly. If the number is not a multiple of 3, 5, or 15, it simply prints the number.

The `Main` action runs a loop from 1 to 100, invoking the `FizzBuzz` action for each number.

```

1  <IsMultiple>[DefineAction]{
2      <Input>{num, base}
3      <Default>"Not a multiple."
4      <Body>{
5          [Condition]{
6              [Equals]{{Modulo}{num, base}, 0},
7              <Return>"Is a multiple.",
8              <Return>"Not a multiple."
9          }
10     }
11 }
12
13 <FizzBuzz>[DefineAction]{
14     <Input>{num}
15     <Default><Error>"An error has occurred."
16     <Body>{
17         [Condition]{
18             [Equals]{{IsMultiple}{num, 15}, "Is a multiple."},
19             [Print]{"FizzBuzz"},
20             [Condition]{
21                 [Equals]{{IsMultiple}{num, 3}, "Is a multiple."},
22                 [Print]{"Fizz"},
23                 [Condition]{
24                     [Equals]{{IsMultiple}{num, 5}, "Is a multiple."},
25                     [Print]{"Buzz"},
26                     [Print]{num}
27                 }
28             }
29         }
30     }
31     <Return>{0}
32 }
33
34 <Main>[DefineAction]{
35     <Input> is {}
36     <Default><Error>"An Error has occurred."
37     <Body>{

```

```

        <num>1
        [While]{
            [LessThan]{num, 101},
            <Body>{
                [FizzBuzz]{num}
                <num>[Add]{num, 1}
            }
        }
        <Return>{0}
    }
}

```

Factorial

This program calculates the factorial of a number. The `Factorial` action checks if the input number is 0 or 1. If it is, it returns 1; otherwise, it multiplies the number by the factorial of (number - 1).

```

1  <Factorial>[DefineAction]{
2      <Input>{num}
3      <Default><Error>"An error has occurred."
4      <Body>{
5          [Condition]{
6              [Or]{{[Equals]{num, 0}, [Equals]{num, 1}},
7                  <Return>1,
8                  <Return>[Multiply]{num, [Factorial]{{[Subtract]{num, 1}}}}
9          }
10     }
11 }
12
13 <Main>[DefineAction]{
14     <Input> is {}
15     <Default><Error>"An error has occurred."
16     <Body>{
17         <num>5
18         <fact>[Factorial]{num}
19         [Print]{fact}
20         <Return>{0}
21     }
22 }

```

Fibonacci Sequence

The Fibonacci Sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1.

```

1  <Fibonacci>[DefineAction]{
2      <Input>{num}
3      <Default><Error>"An error has occurred."
4      <Body>{
5          [Condition]{
6              [Or]{{[Equals]{num, 0}, [Equals]{num, 1}},
7                  <Return>{num},
8                  <Return>[Add]{{[Fibonacci]{{[Subtract]{num, 1}}, [Fibonacci]{{[Subtract]
9 {num, 2}}}}
10         }
11     }
12 }
13

```

```

23
14 <Main>[DefineAction]{
15     <Input> is {}
16     <Default><Error>"An error has occurred."
17     <Body>{
18         <num>10
19         <fibonacci>[Fibonacci]{num}
20         [Print]{fibonacci}
21         <Return>{0}
22     }
23 }
24
25

```

This program generates the Fibonacci number at a specified position in the sequence.

The `Fibonacci` action checks if the input number is 0 or 1. If it is, it returns the number itself (since the first two numbers in the Fibonacci sequence are 0 and 1); otherwise, it calculates the Fibonacci number as the sum of the Fibonacci numbers at positions (number - 1) and (number - 2).

Reverse a String

In this program, the `Reverse` action iterates backwards through the input string, concatenating each character onto a new string `reversed`.

```

1 <Reverse>[DefineAction]{
2     <Input>{str}
3     <Default><Error>"An error has occurred."
4     <Body>{
5         <length>[Length]{str}
6         <reversed>""
7         [index]>[Subtract]{length, 1}
8         [While]{
9             [GreaterThanOrEqual]{index, 0},
10            <Body>{
11                <reversed>[Concatenate]{reversed, [AccessAt]{str, index}}
12                <index>[Subtract]{index, 1}
13            }
14        }
15        <Return>{reversed}
16    }
17 }
18
19 <Main>[DefineAction]{
20     <Input> is {}
21     <Default><Error>"An error has occurred."
22     <Body>{
23         <str>"Hello, World!"
24         <reversed>[Reverse]{str}
25         [Print]{reversed}
26         <Return>{0}
27     }
28 }

```

Regards,
Cora

Appendix: Turing-Completeness Review

A computational system is considered Turing-complete if it can simulate any Turing machine. This means that the system can perform any computation that can be performed by a Turing machine, given enough

time and memory. The qualifiers for Turing-completeness generally include:

1. **Conditionals:** The system must support conditional branching, such as if-then-else constructs, which allow the system to execute different instructions based on certain conditions.
2. **Variables and State:** The system must have a way to store and manipulate information, typically through variables or some form of mutable state.
3. **Loops or Recursion:** The system must be capable of repeating a set of instructions indefinitely, either through loops or recursive function calls.
4. **Input/Output:** The system should be able to interact with some form of input and produce output.
5. **Infinite Loops or Unbounded Recursion:** The system must have the ability to perform unbounded loops or recursion, allowing for the potential of infinite computation, which is a hallmark of Turing machines.
6. **Arbitrary Levels of Indirection/Nesting:** The system should support arbitrary levels of indirection, such as pointers to pointers or nested function calls, to provide the necessary complexity and depth.

Evaluating TACL against these qualifiers:

1. **Conditionals:** TACL has `[Condition]` with `{condition, [ActionIfTrue]{parameters}, [ActionIfFalse]{parameters}}` for if-then-else statements, fulfilling this requirement.
2. **Variables and State:** TACL uses *Things* as variables to store and manipulate data. The language supports assignment and transportation of *Things*, allowing for mutable state.
3. **Loops or Recursion:** TACL includes constructs for both while loops (`[While]`) and for loops (`[For]`). Although recursion isn't explicitly mentioned, the ability to define actions (`[DefineAction]`) implies that functions can be created, which could potentially call themselves, achieving recursion.
4. **Input/Output:** TACL includes the `[Print]` action for output. While input isn't explicitly detailed in the provided specification, the `<Input>` parameter in action definitions suggests the language can accept input.
5. **Infinite Loops or Unbounded Recursion:** Since TACL has while loops, it can theoretically run an infinite loop, assuming there are no built-in stopping conditions or limitations on loop execution.
6. **Arbitrary Levels of Indirection/Nesting:** TACL's ability to define actions, which can take other actions as parameters, and the use of *Boxes* and *Piles* for collections, implies that it supports multiple levels of indirection or nesting.

Based on the provided language specification, TACL fulfills the basic criteria for Turing-completeness. It has control flow constructs, supports mutable state, can perform indefinite computation through loops, and seems to support function definition and invocation, which could allow for recursion. It is also suggested that it can handle input and output. However, without explicit evidence of the language's capability to handle unbounded recursion or to perform arbitrary computation, the claim of Turing-completeness should be considered with a degree of caution. The actual Turing-completeness of TACL would be best proven through implementation and testing, demonstrating that it can indeed simulate any Turing machine.

Regards,
Cora

Appendix: Syntax Completion Surveys

These requests were given to Cora with the task of searching out what needs to be added or considered to the language.

Bubble Sort

Prompt for Reviewer

"Use this language to program a bubble sort. Walk me through the bubble sort algorithm first, then implement it in TACL. Use a Main action as the execution point and as many other actions as needed. Afterward, explain any Actions you had to write from scratch that weren't in the reference, or any language concepts we need to work on to ensure completeness."

Response from Reviewer

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no more swaps are needed, indicating that the list is sorted.

The algorithm can be broken down into the following steps:

1. Start from the beginning of the list.
2. Compare the first element with the second element.
3. If the first element is greater than the second element, they are swapped.
4. The next pair of adjacent elements is compared and the process is repeated until the end of the list.
5. This process continues, making multiple passes through the list until no more swaps are needed.

Here's how we can implement Bubble sort in TACL:

```
1  <Swap>[DefineAction]{
2      <Input>{pile, index1, index2}
3      <Default><Error>"An error has occurred."
4      <Body>{
5          <temp>[AccessAt]{pile, index1}
6          <pile>[ReplaceAt]{pile, index1, [AccessAt]{pile, index2}}
7          <pile>[ReplaceAt]{pile, index2, temp}
8          <Return>{pile}
9      }
10 }
11
12 <BubbleSort>[DefineAction]{
13     <Input>{pile}
14     <Default><Error>"An error has occurred."
15     <Body>{
16         <length>[Length]{pile}
17         <swapped>"true"
18         [While]{
19             [Equals]{swapped, "true"},
20             <Body>{
21                 <swapped>"false"
22                 <i>0
23                 [While]{
24                     [LessThan]{i, [Subtract]{length, 1}},
25                     <Body>{
26                         [Condition]{
27                             [GreaterThan]{[AccessAt]{pile, i},
28 [AccessAt]{pile, [Add]{i, 1}}},
29                             <Body>{
30                                 <pile>[Swap]{pile, i, [Add]{i,
31 1}}
32                                 <swapped>"true"
33                             },
34                             <Body>{}
35                         }
36                         <i>[Add]{i, 1}
37                     }
38                 }
39             }
40         }
41     }
```

```

42         <Return>{pile}
43     }
44 }
45
46 <Main>[DefineAction]{
47     <Input> is {}
48     <Default><Error>"An error has occurred."
49     <Body>{
50         <pile>{(1)4, (2)2, (3)9, (4)6, (5)7, (6)1}
51         <sorted>[BubbleSort]{pile}
52         [Print]{sorted}
53         <Return>{0}
54     }
55 }

```

In this code, we define two new actions `Swap` and `BubbleSort`.

`Swap` is a new action which swaps two elements at the given indexes in a pile. It first saves the element at `index1` into a temporary variable `temp`, then replaces the element at `index1` with the element at `index2`, and finally replaces the element at `index2` with `temp`. It returns the modified pile.

`BubbleSort` is the main action that implements the bubble sort algorithm. It takes a pile as input and sorts it in ascending order. It uses a nested while loop to repeatedly traverse the pile and swap any adjacent elements that are in the wrong order. The outer while loop continues until a complete pass through the pile is made without any swaps, indicating that the pile is sorted.

The `Main` action creates a pile of unsorted numbers, sorts it using `BubbleSort`, prints the sorted pile, and returns 0.

In terms of language completeness, we had to define a `Swap` action from scratch because swapping two elements in a list is a common operation in many algorithms, including sorting algorithms. Also, the `ReplaceAt` action was used, which replaces an element at a specific index in a pile. It's a core operation for manipulating lists or piles in our language. It would be beneficial to include these actions in the standard library of TACL.

Regards,
Cora

Binary Search Tree

Prompt for Reviewer

"Use this language to program a binary search tree. Walk me through the binary search algorithm first, then implement it in TAC. Use a Main action as the execution point and as many other actions as needed. Afterward, explain any Actions you had to write from scratch that weren't in the reference, or any language concepts we need to work on to ensure completeness."

Response from Reviewer

A binary search tree is a data structure that allows for efficient searching, insertion, and deletion of values. The key property of a binary search tree is that for any given node, all nodes in its left subtree have values less than the node's value, and all nodes in its right subtree have values greater than the node's value.

The binary search algorithm is a method for finding a target value within a sorted array. It halves the search space at every step until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Here's the step by step process for binary search algorithm:

1. Compare the target value to the middle element of the array.

2. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found.
3. If the search ends with the remaining half being empty, the target is not in the array.

Now, let's implement this in TACL language.

```

1  <BinarySearchTree>[DefineAction]{
2      <Input>{value, left, right}
3      <Default><Error>"An error has occurred."
4      <Body>{
5          <value>value
6          <left>left
7          <right>right
8          <Return>{value, left, right}
9      }
10 }
11
12 <Insert>[DefineAction]{
13     <Input>{tree, value}
14     <Default><Error>"An error has occurred."
15     <Body>{
16         [Condition]{
17             [Equals]{tree, null},
18             <Return>[BinarySearchTree]{value, null, null},
19             <Body>{
20                 [root, left, right][Split]{tree, 3}
21                 [Condition]{
22                     [LessThan]{value, root},
23                     <left>[Insert]{left, value},
24                     <right>[Insert]{right, value}
25                 }
26                 <Return>[BinarySearchTree]{root, left, right}
27             }
28         }
29     }
30 }
31
32 <Search>[DefineAction]{
33     <Input>{tree, value}
34     <Default>"Value not found."
35     <Body>{
36         [Condition]{
37             [Equals]{tree, null},
38             <Return>"Value not found.",
39             <Body>{
40                 [root, left, right][Split]{tree, 3}
41                 [Condition]{
42                     [Equals]{value, root},
43                     <Return>"Value found.",
44                     <Body>{
45                         [Condition]{
46                             [LessThan]{value, root},
47                             <Return>[Search]{left, value},
48                             <Return>[Search]{right, value}
49                         }
50

```

```

50
51
52
53
54
55
56 }
57
58 <Main>[DefineAction]{
59     <Input> is {}
60     <Default><Error>"An error has occurred."
61     <Body>{
62         <tree>null
63         <values>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
64         <num_values>[Length]{values}
65         <index>0
66         [While]{
67             [LessThan]{index, num_values},
68             <Body>{
69                 <value>[AccessAt]{values, index}
70                 <tree>[Insert]{tree, value}
71                 <index>[Add]{index, 1}
72             }
73         }
74         <search_value>5
75         <result>[Search]{tree, search_value}
76         [Print]{result}
77         <Return>{0}
78     }
79 }

```

We have defined a few Actions from scratch for this implementation, namely `BinarySearchTree`, `Insert`, and `Search`. The `BinarySearchTree` Action is used to create a new binary search tree node. The `Insert` Action is used to insert a value into the binary search tree. The `Search` Action is used to find a value in the binary search tree.

Language Concepts to Improve:

1. **Null Handling:** Currently, we have to explicitly check if the tree is null before calling Actions like `Insert` and `Search`. Adding null safety to the language would improve its robustness and user-friendliness.
2. **OOP Support:** Binary search tree is a classic example where object-oriented programming is beneficial. If we had classes and objects in TACL, we could encapsulate the value, left, and right inside a `BinarySearchTree` class and define `Insert` and `Search` as methods of this class.
3. **Error Handling:** We could improve our error messages by including the value that caused the error and the Action in which the error occurred.

Regards,
Cora

Conway's Game of Life

Prompt for Reviewer

Please write me a program in TACL for Conway's Game of Life. First explain how the Game of Life Works, then write the code in TACL.

Response from Reviewer

Conway's Game of Life is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. It is a cellular automaton invented by mathematician John Horton

Conway. The "game" is a simulation that follows a simple set of rules on a grid of cells, where each cell can be either alive or dead.

Rules of Conway's Game of Life:

1. **Underpopulation:** If a living cell is surrounded by fewer than two living cells, it dies.
2. **Survival:** If a living cell is surrounded by two or three living cells, it survives to the next generation.
3. **Overpopulation:** If a living cell is surrounded by more than three living cells, it dies.
4. **Reproduction:** If a dead cell is surrounded by exactly three living cells, it becomes a live cell.

The game proceeds in discrete time steps, called generations. The entire board is updated at once. The initial pattern constitutes the seed of the system, and the first generation is created by applying the above rules simultaneously to every cell in the seed—live or dead cells are birthed or die for the next generation.

Let's write the program in TACL, assuming we have a grid represented by a Pile of Piles, where each sub-Pile represents a row and each Thing in the Pile represents a cell. True indicates a live cell, and False indicates a dead cell.

```
<UpdateCell>[DefineAction]{
  <Input>{rowIndex, colIndex, grid},
  <Default><error>"Invalid cell coordinates.",
  <Body>{
    <neighbors> is [
      GetNeighbors] with {rowIndex, colIndex, grid}
    <aliveNeighbors>[CountTrue]{neighbors}
    <currentState>[AccessAt]{grid, rowIndex, colIndex}
    <newState>False
    [Condition]{
      [Equals]{aliveNeighbors, 3},
      <newState>True,
      [Condition]{
        [And]{{[Equals]{aliveNeighbors, 2}, currentState},
          <newState>True,
          <newState>False
        }
      }
    }
    <Return>{newState}
  }
}

<GetNeighbors>[DefineAction]{
  <Input>{rowIndex, colIndex, grid},
  <Default><error>"Invalid cell coordinates.",
  <Body>{
    <neighbors><Box>{}
    <maxRows>[Subtract]{{[PileLength]{grid}, 1}
    <maxCols>[Subtract]{{[PileLength]{[AccessAt]{grid, rowIndex}}, 1}
    <StartRow>[Max]{{[Subtract]{rowIndex, 1}, 0}
    <EndRow>[Min]{{[Add]{rowIndex, 1}, <maxRows>}
    <StartCol>[Max]{{[Subtract]{colIndex, 1}, 0}
    <EndCol>[Min]{{[Add]{colIndex, 1}, <maxCols>}
    [For]{
      <i><StartRow>,
      [LessThanOrEqual]{<i>, <EndRow>},
      <i>[Increment]{<i>, 1},
      [For]{
```

```

        <j><StartCol>,
        [LessThanOrEqual]{<j>, <EndCol>},
        <j>[Increment]{<j>, 1},
        [Condition]{
            [Or]{
                [Not]{[Equals]{<i>, rowIndex}},
                [Not]{[Equals]{<j>, colIndex}}
            },
            [AddToBox]{<neighbors>, [AccessAt]{grid, <i>, <j>}},
            {}
        }
    }
}
<Return>{neighbors}
}
}

<CountTrue>[DefineAction]{
    <Input>{box},
    <Default><error>"Input must be a Box.",
    <Body>{
        <count> is 0
        [ForEach]{
            <thing><box>,
            [Condition]{<thing>, <count>[Increment]{<count>, 1}, {}}
        }
        <Return>{<count>}
    }
}

<GameOfLifeStep>[DefineAction]{
    <Input>{grid},
    <Default><error>"Input must be a grid.",
    <Body>{
        <newGrid><Box>{}
        <maxRows>[Subtract]{[PileLength]{grid}, 1}
        <maxCols>[Subtract]{[PileLength]{[AccessAt]{grid, 0}}, 1}
        [For]{
            <rowIndex>0,
            [LessThanOrEqual]{<rowIndex>, <maxRows>},
            <rowIndex>[Increment]{<rowIndex>, 1},
            [For]{
                <colIndex>0,
                [LessThanOrEqual]{<colIndex>, <maxCols>},
                <colIndex>[Increment]{<colIndex>, 1},
                <newState>[UpdateCell]{<rowIndex>, <colIndex>, <grid>}
                [AddToPile]{[AccessAt]{<newGrid>, <rowIndex>}, <newState>}
            }
        }
        <Return>{<newGrid>}
    }
}

<Main>[DefineAction]{
    <Input>{},
    <Default>"Game of Life ended unexpectedly.",
    <Body>{

```

```

    <grid>{
      {(1)False, False, True, False, False}
      {(2)False, False, True, False, False}
      {(3)False, False, True, False, False}
      {(4)False, False, False, False, False}
      {(5)False, False, False, False, False}
    }
    <iterations>5
    [For]{
      <i>0,
      [LessThan]{i, <iterations>},
      <i>[Increment]{i, 1},
      <grid>[GameOfLifeStep]{grid}
      [Print]{grid}
    }
    <Return>{0}
  }
}

```

In this code, `GetNeighbors` retrieves the neighboring cells of a given cell, `CountTrue` counts how many of those neighbors are alive, `UpdateCell` decides the fate of a cell based on the rules of the game, and `GameOfLifeStep` updates the entire grid for a single generation. The `Main` action initializes the game with a predefined grid and runs through a set number of iterations, printing the grid at each step.

Please note that this is a conceptual representation of Conway's Game of Life in TACL. Some actions like `[ForEach]`, `[AddToBox]`, `[AddToPile]`, `[PileLength]`, `[Max]`, and `[Min]` are assumed to exist for the purpose of this exercise, and the code structure follows the specification provided earlier.

Regards,
Cora

Appendix: "Syntactic Sugar"

In TACL, syntactic sugar can be found in various places to make the syntax more concise and to improve readability. Here are some examples:

1. Concise Assignment:

- Plain form: `<Assignment> is 5`
- Concise form: `<Assignment>5`

2. Concise Action Invocation:

- Plain form: `<Result> is [Action] with {param1, param2, ...}`
- Concise form: `<Result>[Action]{param1, param2, ...}`

3. Concise Action with Transport:

- Plain form: `[Action] with {param1, param2, ...}`
- Concise form: `[Action]{param1, param2, ...}`

4. Concise Box and Pile Initialization:

- Plain form for Box: `<myBox> is {Thing1, Thing2, Thing3}`
- Concise form for Box: `<myBox>{Thing1, Thing2, Thing3}`
- Plain form for Pile: `<myPile> is {Thing1 at 1, Thing2 at 2, Thing3 at 3}`
- Concise form for Pile: `<myPile>{(1)Thing1, (2)Thing2, (3)Thing3}`

5. Concise Control Flow:

- If-Then-Else Plain form: `[Condition] with {condition, [ActionIfTrue]{params}, [ActionIfFalse]{params}}`
- If-Then-Else Concise form: `[Condition]{condition, [ActionIfTrue]{params}, [ActionIfFalse]{params}}`
- While Loop Plain form: `[While] with {[ConditionAction]{params}, [ActionSequence]{params}}`
- While Loop Concise form: `[While]{[ConditionAction]{params}, [ActionSequence]{params}}`
- For Loop Plain form: `[For] with {[InitializationAction]{params}, [ConditionAction]{params}, [UpdateAction]{params},`

```
[ActionSequence]{params}}
```

- For Loop Concise form: `[For]{[InitializationAction]{params}, [ConditionAction]{params}, [UpdateAction]{params}, [ActionSequence]{params}}`

6. Omission of Commas:

- When a Box or Pile contents are fully separated by lines, commas can be omitted:

```
<myBox>{  
  Thing1  
  Thing2  
  Thing3  
}
```

Instead of:

```
<myBox>{Thing1, Thing2, Thing3}
```

7. Index Context Simplification:

- Instead of using 'at' for indices in a Pile in plain form, you can use parentheses:
 - Plain form: `Thing1 at 1`
 - Concise form: `(1)Thing1`

Each of these examples of syntactic sugar in TACL allows for a more streamlined way of writing the same instruction, catering to different preferences of verbosity and readability.

Regards,
Cora

Appendix: Paradigms

The Thing-Action-Context Language (TACL) specification suggests a multi-paradigm approach. Below are the paradigms that TACL seems to follow, based on the syntax and features described in the specification:

1. **Imperative Programming:** TACL syntax specifies explicit sequences of commands for the computer to perform. This is evident from the control flow constructs (`[For]`, `[While]`, `[Condition]`) and the action invocation syntax (`<Result>[Action]{param1, param2, ...}`), which directly manipulate program state.
2. **Procedural Programming:** As a subset of imperative programming, TACL supports procedures or actions (`[DefineAction]`) that encapsulate a sequence of steps, receive input parameters, and return a result. This is analogous to functions or procedures in other procedural languages.
3. **Object-Oriented Programming (OOP):** While TACL does not explicitly mention classes or objects, the concept of "Things" as entities that can encapsulate data and actions (methods) suggests an object-like structure. However, the language specification does not provide details on inheritance or polymorphism, which are hallmarks of OOP, so it may only partially implement this paradigm.
4. **Declarative Programming:** This is less apparent, but the specification's emphasis on readability and using constructs that resemble natural language could be interpreted as a nod to declarative programming. Declarative languages focus on the "what" rather than the "how" of problem-solving. TACL's syntax for defining actions and handling errors (`[Attempt]`) abstracts away some of the operational details, focusing on the outcomes rather than the process.
5. **Functional Programming:** There is no explicit mention of functional programming features such as first-class functions, immutability, or pure functions. However, the action definitions and invocations resemble the use of functions in functional languages. Without further details on how state is managed or whether functions are first-class citizens in TACL, it's difficult to determine the extent to which it adopts functional programming principles.
6. **Event-Driven Programming:** TACL's error handling mechanism (`[Attempt]`) suggests some degree of event-driven programming where actions are taken in response to events (in this case, exceptions). However, there is no mention of event loops or event handlers, which are common in event-driven systems.

In conclusion, TACL appears to most strongly align with imperative and procedural programming paradigms, as it emphasizes explicit sequences of operations through actions (procedures/functions) and state manipulation. It has object-oriented tendencies with its concept of "Things" but lacks a full suite of OOP features based on the provided specification. The declarative aspect is hinted at through its readable syntax but is not a primary focus. Functional programming elements are not evident from the specification. Finally, while there is some element of event-driven programming in the error handling, this alone does not classify TACL as an event-driven language.

Regards, Cora

END AI-GENERATED CONTENT