

Augury - Final Report

Brett Challice, Virya Shields

April 13, 2024

Abstract

Augury is a specialized programming language created for developing programs that execute sequences of game-related actions across various platforms, ultimately aiming to enhance game-testing procedures upon mastery. This piece explores the unique features and aspects that distinguish Augury from other programming languages, particularly in terms of its application and usefulness within its specific domain. The report primarily focuses on the process of selecting design pathways for the language to achieve its intended goal, which is a challenging, yet inherent aspect of program language innovation.

1 Motivation

Augury's main objective is to incorporate and adjust certain aspects of object-oriented paradigms, event handling, and concurrency to enhance the development of player simulations. Although comparable programs can be developed using other languages, our emphasis lies in customizing Augury precisely for this task. This approach offers a unique advantage over other languages in this domain, capitalizing on the ease and speed of development. The secondary motivator behind the creation of Augury is the difficulty of learning modern languages; all seemingly having nuances that make them dissimilar from other languages, necessitating vigorous research and trial and error to hone. Augury on the other hand utilizes existing syntactical and semantic structures prevalent in ubiquitous languages that are commonly some of the first languages programmers gain experience in. Thus, with just a short amount of time, Augury can feel intuitive. Even with minimal experience; Augury abstracts programming principles at a high level on top of having simplistic syntax, making it the ideal stepping stone to other languages. In essence, Augury strives to lower the entry barrier for new programmers, making the learning of player automation both engaging and functionally adaptable. This enables users to create complex simulations that closely mimic real players interacting within a game environment, without requiring an immense amount of time to learn.

1.1 Comparison with other existing languages

Augury has its foundation in Python and the C family of programming languages, drawing upon their object-oriented nature while also incorporating some of their distinctive features. The preference for these languages stems from their widespread usage and teaching in general-purpose programming. As a result, the creators of Augury have naturally gravitated towards them. In comparison to Python, Augury uses white-space delimitation to separate elements within pieces of code, eliminating the need for statement terminators such as semicolons. Augury uses dynamic typing, demanding the need for an interpreter to manage the types using runtime descriptors. The iterative statements also bear a strong resemblance, being concise and aligning well with the ease of implementation. Finally, when initializing variables in a constructor, the process mirrors that of Python. However, unlike in Python where "self" is included both as a formal parameter and part of the assignment, in our language, we use "this" exclusively as part of the assignment. In Python, "self" is employed in formal parameters to distinguish instance methods from static methods, but in our approach, all elements within a "Player" construct are related to instances only, making the use of "this" unnecessary in formal parameters. Static elements are handled separately in a "Helper" construct, marking a significant paradigm shift. This distinction will be elaborated upon and underscored later in the paper. Relating to the C family, particularly C#, Augury utilizes "sections" instead of namespaces. These sections serve as encapsulation tools, effectively preventing naming conflicts in scenarios where multiple programmers

collaborate on the code simultaneously, identical to the functionality in C#. Augury uses identical commenting schemes as the C family, for single-line and multi-line comments. Access modifiers are similar to that of the C family, although the defaults contrast. In Augury, we prioritized usability over reliability by defaulting all modifiers to “public”. This decision acknowledges that user task automation typically doesn’t prioritize security as a primary concern. Augury employs an event-handling syntax that closely resembles the exception-handling syntax found in the C family of languages. Finally, Augury’s implementation of concurrency resembles much of the same concurrency model as that in C. Throughout the rest of the report, it’s evident how these languages have influenced the syntactical and semantic structure of Augury, a point that is consistently highlighted.

1.2 Intended Audience

The intended audience is multi-fold. The central application is intended for game developers, as they do not typically have the time to thoroughly troubleshoot the games themselves, indicating the necessity to hire and pay game testers. Augury allows the programmers to simulate player interaction through code, gaining all the necessary information during the development cycle and the release phases before the game is launched; without having to fork out the money to fill the testing position. Using Augury, these developers can simulate multiple players of varying behaviors to test games for glitches or unexpected results themselves. The secondary audience consists of novice programmers interested in learning object-oriented programming. Augury abstracts a plethora of various concepts that are typically challenging to grasp for newcomers. With only a few language-specific features that require knowledge, beginners can create simplistic programs that can still be quite powerful. Augury caters to individuals with diverse technological backgrounds, ranging from high to low expertise. It abstracts just enough to facilitate easy implementation while providing essential exposure to fundamental concepts that can be applied to similar languages in the future. Augury also employs an enjoyable approach to learning programming, emphasizing practical application over pure computation. This approach is crucial for engaging novice programmers and maintaining their interest. In many programming learning methods, challenges are often limited to tasks within the program, such as solving arithmetic problems to achieve a specific output. Augury, however, empowers new programmers to create code that interacts with the broader environment, moving beyond the confines of the program itself. This immersive approach not only fosters deeper engagement but also illustrates how programming extends beyond simple calculation, offering insight into the realm of system calls as an educational avenue. The tertiary audience includes the game owners. For established games releasing occasional patches, they can offer APIs for internal libraries to abstract implementation to Augury developers, ultimately enabling quicker testing and subsequent patch delivery to users. An example provided in the Appendix is a partial Runescape implementation for automating tasks such as MiningPlayer and FishingPlayer. Initially, the program requires a method to identify specific in-game objects to execute actions on them. Runescape could provide a range of resources for Augury’s use, potentially offering them for a fee. This would prevent programmers from relying solely on coordinate systems or having to develop color/object recognition algorithms themselves, which can be extremely tedious and scales with how intricate the game is. These resources might include comprehensive documentation, pre-built libraries, and specialized tools tailored to Runescape’s game environment, streamlining the adaptation process for programming tasks specific to the game.

2 Innovative Features

2.1 Players & Helpers

2.1.1 Motivation

Object-oriented languages rely on mechanisms to encapsulate data and methods into a single distinct unit called a class. Classes can be of two types - static and non-static. Non-static classes can be instantiated into object instances, and static classes serve as libraries of code, aiding the program by offering methods. Augury differentiates these types by introducing two new encapsulation entities - Players and Helpers. The rationale behind this choice is to simplify the concept for novice programmers who may find it challenging to grasp the distinction between entities that are static versus non-static. The term “static” can be used in various contexts, leading to confusion - for example, comparing a

static class to a static variable. A static class means it cannot be instantiated, whereas a static variable means it belongs to the class itself. Moreover, since the term can be daunting, we abstracted it to enhance comprehension.

2.1.2 General Schema

Augury does provide support for classes, but they are secondary and typically utilized for defining objects that assist in sequencing, as sequencing is Augury’s main specialty. However, there’s a requirement in Augury regarding how programs commence. Similar to the C family, every Augury program must include a `main()` function to initiate program execution, which, in Augury, is situated within a class. This is the only mandatory class - the one that contains the main function. Continuing on Players and Helpers in Augury, a Player is akin to a non-static class, whereas a helper resembles a static class. Typically, both types of classes can incorporate static or non-static properties, methods, and variables. However, in Augury, Players exclusively accommodate instance entities, while helpers exclusively contain static entities. Players manage everything pertaining to instances, while helpers manage everything specific to the Player construct itself. Consequently, for Players to include static data members and methods, they must be associated with a helper to oversee these elements. The helper must be linked to a sequence to fulfill this role. The way to achieve this is to include the keyword “binds” in the helper definition, indicating which Player definitions the helper is bound to, by sequence name. Players and helpers are vital elements of the language, collaborating seamlessly to accomplish the same functions as a class, but with greater intuitiveness tailored to Augury’s domain.

2.1.3 Players

The Player entity is an encapsulation construct that resembles a non-static class but exclusively accommodates instance operations and variables. All elements within a Player construct pertain directly to the Player itself, and instances of the Player can utilize the provided definitions as needed. Whenever a Player construct is instantiated into a Player instance, there are a few fundamental rules that are followed. Several fundamental rules come into play when a Player construct is instantiated as a Player instance. Initially, the players run in their respective threads, which are automatically and implicitly created upon instantiation within the main method. An embedded function exists for each sequence to start, the `start()` function, which is called in main to make a player instance start execution. Every Player construct requires an asynchronous `engage()` sequence, serving as the entry point once an instance is started. The engage sequence is akin to a main function, but acts within the Player instance itself, controlling the order of execution, calling functions, and other sequences. The `engage()` function can optionally be passed arguments. Players support inheritance, as well as the creation of generic Player constructs that are only meant to be inherited from, and overridden. To resolve the issue of multiple inheritance of Players, interfaces can be employed. However, if inheriting from only one “super player,” abstract Player constructs are the preferred approach. The inheritance syntax follows the structure: `{Player} inherits {SuperPlayer}`.

```
Player APlayer >>

:: APlayer() >>
    this.variable = x

async sequence engage() >>

    // Implementation
```

Figure 1: Minimal requirements to create a Player construct

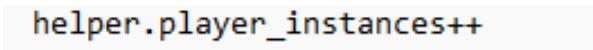
2.1.4 Helpers

Helpers are encapsulation constructs that are closely related to that of a static class in other languages. All functions that reside in Helpers are exclusively static. They are not instantiated, they are isolated

and serve the purpose of binding to Player types, handling their static variables, and providing static operations to aid in execution. Helpers can bind to multiple player types, in which all instances of those player types can access the operations present in the Helper. Once a Helper is bound to a Player construct, the Player code can reference the Helper's variables and functions in one of two ways. The entire Helper handle can be explicitly referenced and targeted using the accessing operator (`.`), or a shorthand reference can be used by explicitly saying "helper.[method/variable]". This works due to the Helper-Player association existing at a many-to-one ratio; Helpers can be bound to one or more Player types, but Player types can only have one Helper bound to them at a given time. Regarding concurrency and the prevention of race conflicts, Helpers also manage shared resources that instances interact with, a topic that will be explored further later on, hence the use of "shared" in the following code. Helpers are implicit monitors that govern mutual exclusion, thereby responsible for competitive synchronization. Here is an example helper construct, present in our complete program found in the Appendix.

```
helper PlayerVariables binds MiningPlayer, FishingPlayer >>

// Static variables
shared player_instances = 0
shared net_mining_profit = 0
shared net_fishing_profit = 0
shared net_total_profit = 0
```



```
helper.player_instances++
```

Figure 2: Shorthand helper reference in a bound Player

2.2 Sequences & Functions

2.2.1 Sequences

Sequences are callable entities nested solely in Player constructs. They are closely related to functions concerning syntax, although they are separate in the sense that their purpose is to perform an action within the game they are programmed to interact with. Rather, they resemble procedures that deploy a set of actions. Sequences are strictly instance-based, as only instances of players intuitively can perform a sequence of actions, not the Player construct itself. Sequences can nest other sequence calls, as well as functions. Due to the concurrent aspect of Augury and given the natural interaction of game players, most of the sequences are expected to be asynchronous. They're termed sequences not because they run sequentially, but as a reminder that their purpose is to carry out a list of actions rather than to evaluate. Sequences are the backbone of Augury's domain, as they are the only vector for interacting with the underlying game application.

```
// Example sequences (Async & Sync)

async sequence example()
sequence example(arguments)
```

2.2.2 Functions

In the context of functions, functions serve as a unit of evaluation that can return results to callers. They are the same as any other language, able to optionally take input, evaluate, and optionally return output. Functions ultimately act to complement sequences and provide results to aid in sequence execution. Like sequences, functions can also execute asynchronously.

2.2.3 Similarities/Differences

Sequences and functions share more similarities than differences. Their syntax is nearly identical, differing only in the keywords “sequence” and “function” in their respective headers. Both can be optionally asynchronous and can accept input arguments. The core differences between sequences and functions lie in whether they can return values, and whether they can directly interact with the game application. Functions are capable of returning results, while sequences cannot return anything. However, sequences have the unique ability to interact with the game environment, which functions cannot. Essentially, functions cannot make system calls, whereas sequences possess this capability. Functions also cannot contain event handlers, but sequences can. Functions can exist within classes and helpers, sequences however cannot. This differentiation serves to clarify the distinct purposes of both entities, providing a clear understanding of which directly impacts Player actions, versus what aids Player actions, and aids in organizational clarity.

2.3 Access Modifiers

Besides the main function, all other constructs in Augury have the default “public” access modifier. Certainly, the default settings can be changed by explicitly specifying a modifier from the list of modifiers: protected, private, public, or internal, similar to the conventions found in the C family of languages. This style was chosen to prioritize ease of implementation, making all contents accessible to the user, rather than emphasizing the importance of reliability and minimizing dependencies, which takes excess time. Game developers have limited time to allocate to deciding what to expose, as this is a secondary responsibility alongside game development. Therefore, implementation must be quick, even if it means disregarding certain reliability or security mechanisms. Furthermore, we operate under the assumption that the code is not intended for end users; rather, the developer who creates it will be the one executing it. In such a controlled environment, where the developer is the user, prioritizing security is not considered at the forefront of our design concerns.

2.4 Data Types

Augury introduces several new predefined data types that are uncommon in other programming languages, mostly to aid in object detection and recognition in the game client, with one to aid in code writing.

2.4.1 2D & 3D Vectors

The first type, which is a primitive data structure and a subset of an array in other languages, includes 2D and 3D vectors. These vectors are particularly intuitive in game applications because they can be utilized in coordinate-based algorithms, enabling precise pointer/client control positioning and movement in both two-dimensional and three-dimensional environments. They can be initialized by passing numeric values concerning the axis, nested in curly braces ({}) to distinguish a vector from a list.

```
coord2D = {10,50}    // x ,y
coord3D = {20,25,50} // x,y,z
```

2.4.2 Color Type

The second is a primitive color type, that can be initialized with a hexadecimal value, or a 3d vector representing RGB values. An example below:

```
bright_blue = #0096FF
cobalt_blue = rgb({0, 71, 171})
```

While these two types can function independently, their utility is somewhat restricted. A more potent application involves incorporating them into lists and initializing them within Player constructs, thereby providing valuable data for use in their sequences.

2.4.3 Action List

The third and last introduction is not a “new” type per-say, but a utilization of lists. Lists can contain any primitive type, but an extension in Augury is lists that contain actions - coined Action lists. Action lists are lists composed of sequences and functions to be executed one after another, from first to last determined through assignment. The ordering only matters for asynchronous functions that depend on the previous functions to execute, which are handled using `async/await` in the function definitions themselves; the list is not responsible. This opens an avenue of concurrency issues, but the list itself minimizes writing code to execute every function and can be thought of as its mini-sequence of tasks, prioritizing workflow over reliability. Once correctly created, the variable can be called like a function to execute the first function in the Action list. If the entity is a function inside the list and it returns something, the return value is discarded. Action lists are best used for sequences and procedures.

```
actions = [first(), second(), third()]
actions()
```

2.5 Event Handling

Event handling is another essential and uniquely implemented feature in Augury. We use `catch` statements to catch certain events that can happen in sequences, very similar to how the C family uses `catch` statements to handle exceptions. In Augury, there are no pre-defined exceptions nor exception handling at all. All exceptions end the program and inform the user of what happened outside of runtime. Thus, this syntax is open for use in event handling and is quite intuitive semantically to handle events. Augury disregards the use of “try” and “finally”, working only with “throw” and “catch” to throw and catch events. The fundamental mechanism behind event handlers operates through asynchronous processes. Each event handler possesses a thread that consistently awaits a designated event to occur, enabling non-sequential triggering. The program can encounter a result that triggers an event to be thrown, and handled in its corresponding event handler, matching by name. Handlers can be passed arguments to be used in further instructions in response to an event. Once an event is caught and handled, the operation is atomic, meaning all other threads that could exist in asynchronous code are blocked until the event is handled. Once the event is handled, the user has one of two choices to decide on further execution: to restart the sequence or resume code the line after the event handler that caught the event. The keyword “*continue*” needs to be explicitly stated to reiterate the sequence, while by default the code resumes one line after and is implicit. Event handling can only take place within sequences, it cannot be handled in functions - another distinguishing factor of sequences and functions. Handling events in this manner is more succinct compared to creating distinct functions such as “OnClick” in C#. Using `catch` blocks for handling events makes programming more concise and intuitive, than the alternative to event handling in other languages. Event handling plays a pivotal role in the concurrent nature of video games, especially where events occur at unpredictable intervals. In such dynamic environments, efficient event-handling mechanisms are essential for managing the diverse and often spontaneous occurrences within the game confines.

```
// Event Throwing
if(!verify_ore(item)) >>
    throw wrong_item(item)
else >>
    this.inventory.items++

if(!click_model_once(${type}.png)) >>
    throw fish_spot_not_found

if(inventory.check_if_full()) >>
    throw inventory_full
//-----
// Event Handlers
catch(wrong_item(item)) >>
    drop_item(item)
    continue
```

```
catch(inventory_full) >>
    sell()

catch(fish_spot_not_found) >>
    go_to_mining_spot()
    continue
```

2.6 Concurrency

Concurrency plays a crucial role in game development. It is indispensable for ensuring that multiple actions and processes within the game can occur simultaneously. Therefore, it's imperative to integrate fundamental tools for managing synchronization and addressing the associated challenges it presents.

2.6.1 Functional Cooperative Synchronization

The first implementation type of synchronization occurs at the function or sequence level. These functional counterparts operate sequentially and block by default, implying that they execute one instruction at a time until the function finishes, then proceed to the next one, and continue in this manner. Augury manages functional synchronization using `async/await`, akin to JavaScript, but without necessitating promises. Entities designated with the “`async`” keyword in the function or sequence header execute their instructions asynchronously, spawning a new thread for each call, and progressing to the subsequent instruction without pausing for the preceding one to conclude. In certain scenarios, when an asynchronously executing function requires the preceding function's results before proceeding to the next instruction, the use of “`await`” becomes relevant. The sequence or function operates asynchronously for independent instructions, while also having the capability to halt execution when necessary. Awaiting an entity entails waiting until its execution is completed, potentially relying on a result from the said entity. Here's an example utilizing `async/await`. The output may be non-deterministic when `await` is incorrectly used, while it becomes deterministic when `await` is used correctly.

```
// Without Await
function print_line() >>
    print("Inside the function call")

async sequence engage() >>
    print("Before the function call")
    print_line()                // Non-Blocking
    print("After the function call")

// Non-deterministic Output:
Before the function call
After the function call
Inside the function call
//-----
// With Await
function print_line() >>
    print("Inside the function call")

async sequence engage() >>
    print("Before the function call")
    await print_line()          // Blocks execution
    print("After the function call")

// Deterministic Output:
Before the function call
Inside the function call
After the function call
```

2.6.2 Multi-thread Cooperative Synchronization

The second type manages concurrency at the thread level. When a function or sequence requires the creation of multiple threads for optimized execution or specific functionalities, Augury ensures thread synchronization using semaphores. Semaphores are widely used for synchronization in the C family of languages. Augury employs a syntax similar to POSIX semaphores, indicating atomic functions with `wait()` and `post()`. Most concurrency challenges can be addressed using semaphores, monitors, and `async/await`. However, in C, there's a scenario involving mutex conditions and signaling that we're still deliberating whether to implement. In further developments of Augury, we may choose to implement such functionality - to have a thread sleep inside a mutex/semaphore allowing another thread to enter to eventually signal the sleeping thread. In the context of thread communication, we chose to have the threads communicate via non-local variables, whether that be global variables or resources within a Helper construct.

```
// Semaphore Demo
// Creates a semaphore and a list of 3 threads to be used in the current function
async sequence multi_thread() >>
  semaphore = semaphore(1)
  threads = [thread(), thread(), thread()]
  foreach(thread in threads) >>
    thread.start([Sequence|Function])

    semaphore.wait()
    // Critical Code
    semaphore.post()
```

2.6.3 Competitive Synchronization

Augury implements competitive synchronization using Helpers, which are entities containing shared static resources among instances of a Player type. These Helpers also extend their functionality across different Player types they are bound to. Helpers operate akin to Monitors in Java, ensuring that race conditions are avoided when multiple threads try to alter a shared resource simultaneously. When a particular thread accesses a resource labeled as “shared” within a Helper, it obtains exclusive access. Within a Helper, any resource can be explicitly designated as “shared” during declaration, indicating which resources have a mutex lock associated with them. By default, all resources are unlocked unless explicitly declared as shared.

```
// Competitive Synchronization

helper Resources binds Player1, Player2 >>

  shared player_count = 0          // Critical Sections -> Mutex
  shared function mutex_function() >>

    function non_mutex_fuction() >> // No mutex
```

The mechanisms to handle such synchronization are Helpers being implicit monitors, to handle competitive synchronization, while cooperative is handled by `async/await` functions similar to JavaScript.

3 Syntactical & Semantical Structure

As previously mentioned in the introduction to Augury it inherits characteristics from Python and the C family, the structure of Augury mimics that of its ancestors. Going down to the smallest unit some examples of lexemes would be ‘=’, ‘if’ ‘else’, and ‘forever’ (a special keyword that continues an action forever). The sentence structure of Augury is similar to that of Python for example:

```
:: GenericPlayer() >>
  this.game = "Runescape"
```

```
this.platform = "PC"
this.task = "Something generic"
this.username = "Username"
this.store_location = "Some location"
this.inventory = Inventory()
```

In this snippet of code the user is adjusting the constructor to accommodate the game in question Runescape, the “.” lexeme denotes the constructor for the entity, the “>>” denotes where the entity starts or where the language knows to look for when a “generic player” is created which can be seen in the code when a player creates a “mining player”, this is also referred to as the binding. In Augury a full sentence would be ‘ this.game = “Runescape” ’, unlike the c family of languages to pass processing to the next line the code does not need a statement terminator ie “;” this was chosen to save time for coders as well as avoid the curse of the missing semicolon, by avoiding it (the semicolon at the end of sentences) altogether the user of the language does not need to worry if they forgot a punctuation mark 50 lines back. As visible by the structure of the code it also follows a similar paragraph structure to Python, in other words blocks of code are delineated using indentation.

3.1 Variables

Variables in Augury are dynamically bound to type as dynamic binding offers more flexibility in code, dynamic binding also allows newer programmers to easily code without worrying about stating the correct data types at the exact correct moment. Dynamic binding in Augury also follows from the idea of keeping the language fast to write, similarly to Python a simple “x = 3” is a valid declaration of a variable and allows the user to circumvent having to type: “int x = 3;”. Augury overall has variables that fall into 3 types: global, which are accessible throughout the whole program; instances, such as in the constructors; and static, which are found outside of the constructors in a class (class variables). Examples of variable types:

- int, float, double
- Action list
- 2D,3D Vector
- char
- colour
- string
- list
- bool

When it comes to the six main characteristics of variables in relation to Augury they can be summarised as: name, keywords in the language will have the form “player” (xxxx) the Augury keywords are lowercase, the language is not case sensitive for user created aspects, so Player1 and player1 will refer to the same entity, the names of things do not need a prefixes so as to not confuse those who are new to coding; the address of variables is passed by reference and aliasing is allowed, the type of the variable can be any of the aforementioned types and numbers range from -2147483648 through 2147483647 such as in python; the value is determined at runtime as the language dynamically allocates the memory; the lifetime of the variables depends on where they lie in the code, as Augury is not meant for explicit memory allocation the language itself handles the life of the variable as well as the de-allocation when the variable is no longer being used; and finally the scope of the variables are relative to the block of code in which they lie unless the “global” keyword is used.

3.2 Expressions

Augury follows the operator precedence of parentheses, and unary operators, then all numeric calculations are processed as they appear in the line. The logic behind the left-to-right mathematical operations was to ensure that someone can type the code as they think it out, “add two plus two then times by ten” will be x = 2+2*10 rather than having to type x = (2+2)*10, this was done for two reasons, one to make it easier for mathematically challenged people not to have to worry about remembering precedence but also to allow people to type as they logically think out the line, and

also to save time on typing and to allow flexibility in code (though should a user desire they can use parentheses to force a higher level of precedence).

3.3 Operator Precedence & Order of Evaluation

3.3.1 Arithmetic

In Augury, there is no arithmetic operator precedence. Usually, there's a structured hierarchy in place, often adhering to mathematical principles such as BEDMAS (Brackets, Exponents, Division and Multiplication, Addition and Subtraction) in terms of order of evaluation. However, the final decision on the order of operations rests with the creators. Instead of creating an exhaustive list of operator precedence, we established a left-to-right evaluation rule. Instead of providing the expression with the evaluation applied according to the rules of BEDMAS, users can arrange the expression in the order they wish to evaluate it. This rule falls within Augury's scope as it caters to users who may lack a mathematical background and eliminates the need for users to remember precedence rules.

To demonstrate, the following equations are equivalent:

$$a = b + c * d$$

$$a = (b + c) * d$$

3.3.2 Categorical Operator Precedence

Besides arithmetic operations, an expression requires a hierarchy of evaluations to ensure correct evaluation.

Parenthesis-> Unary -> Arithmetic -> Relational -> Boolean -> Assignment

3.4 Coercion

Augury has an interesting take on how to handle coercion within expressions. In implicit conversions, the data type on the leftmost side is considered the target type, and then the remaining variables to the right are attempted to be coerced. If any of the variables listed cannot be converted into a compatible type associated with the target type, the expression will terminate, and the program will exit, displaying the corresponding error message.

```
// Target type: int
int_value = 10 + "b" + 20.2 // Result: 10 + 98 (ASCII for b) + 20 = 128

// Target type: string
string_value = "brett" + 5 // Result: "brett5" ("brett" + "5") - String concatenation

// Target type: float
error = 10.43 + "brett" // Result: Runtime error => Program fault
```

3.5 Statements

When it comes to Augury statements follow the style of:

```
<target_var> <assign_operator> <expression>
E.g
x = 4100
print(x)
(4100)
```

This would be the form for any basic statement, for multi-line blocks of code Augury just requires the user to indent the line, if a user wishes to nest functionalities all they are required to do is to increase the indent (such as in the case of multiple nested "if" statements). Augury supports pre and

post-fix operators such as in the C family of languages, when it comes to the resolution of a statement it is resolved left to right. Overloaded operators are allowed, for example, a user may want to have a generic action that can be made more specific upon use of a specified subclass: `sell_item()` that sells any item in the inventory can be overloaded to specifically sell ore if using a mining instance. There is one case where overloading may be redundant: the comparison of two objects, Augury internally has the comparison operator (written as “==”) which can be used to compare objects or characteristics, for example a user could check if

```
if (this.store_location == location_Y) >>
    sell_item()
else
    go_to_store_location()
```

In this snippet, a user is checking to see if the player is in the correct location to sell items and if not to go to the correct location. To allow for ease of coding Augury also allows for compound assignment (`a = a + b`, can be written as `a += b`). Augury also has support for short-hand function calls such as in:

```
introduceSelf() => go_to_mining_spot()
```

Wherein the “=>” denotes to the system that they are actions to be chained together, it is important to note that as Augury supports the comparison operators there is a big difference between “<=” & “>=” (less than or equal to & greater than or equal to) and “=>” which chains the actions. Finally to keep with the concept of simplicity if a user is assigning or passing multiple variables would just have to ensure that they are in the order they appear:

```
a, b, c = 1, 2, 3
print(a, b, c)
(1, 2, 3)
```

3.6 Functions

As mentioned in the introduction, functions in Augury are similar in syntactical structure to other languages (such as Python) but Augury runs on the idea that it concerns the sequences and players. During the process, the user will rely on sequences to complete the chaining of actions but they do not return results, in case of needing feedback from the system a user of Augury could create a function to work in tandem with a sequence or helper. Example of function syntax:

```
class Starter >>

    function main() >>

        mining = MiningPlayer()
        fishing = FishingPlayer()

        // Calling start() on a player implicitly creates a new thread to handle player
        // execution
        // This new thread initiates the special sequence - engage()
        mining.start()
        fishing.start()

        mining.wait()
        fishing.wait()
```

In this example the function is the “main” part of the program, then the “>>” binds to the actions that the user wants the language to process. From the declaration of `main()` the user is then specifying the type of player they are wishing to create. Upon use of `mining.start()` the information is received

from the inherited `GenericPlayer` and `MiningPlayer` classes and thus the program knows what set of actions each 'player' can complete. The syntax is similar to that of other object-oriented languages as it follows the `thing.action()` format from C languages while using the indentation to make the code faster to read and write.

4 Appendix

4.1 Complete Program Example

```
//-----
/*
  Brett Challice
  4/5/2024
  Augury Demonstration

  Program Description:

      This is a simple program that creates two player instances - a
      MiningPlayer, and a FishingPlayer. The program demonstrates Player
      instances, sequences, and helpers that showcase the backbone of
      Augury's domain. The instances are started, and their engage()
      sequences execute within their implicit threads. The task of the
      players is to navigate to their designated spot to do their task,
      which is to simply click on a node, and gain items until their
      inventory is full, and then when sell the items at a store location.
      Concurrency is handled using async await, similar to JavaScript, and
      event handlers are handled like catch statements identical to how the
      C family handles exceptions.

*/
//-----

// Globals
global usernames_path = "./assets/usernames.txt"
global fish_models = "./assets/fishing/fishmodels"
global ore_models = "./assets/mining/oremodels"

//-----
section Driver >>

    // Contains the entry point to the program
    // Instantiates the players

class Starter >>

    function main() >>

        mining = MiningPlayer()
        fishing = FishingPlayer()

        // Calling start() on a player implicitly creates a new thread to handle player
        // execution
        // This new thread initiates the special sequence - engage()
        mining.start()
        fishing.start()

        mining.wait()
```

```

        fishing.wait()
//-----
section Players >>

// Superclass Generic Player
// Abstract => Cannot be instantiated, can only be inherited from, can contain
// declarations and definitions

abstract player GenericPlayer >>

:: GenericPlayer() >>
    this.game = "Runescape"
    this.platform = "PC"
    this.task = "Something generic"
    this.username = "Username"
    this.store_location = "Some location"
    this.inventory = Inventory()

// Async function introduce_self()
// Async functions do not block further execution, the function can execute
// asynchronously
// The function creates a new thread to complete the task, while the main thread
// continues
async sequence introduce_self() >>
    client.type("${this.username} doing {this.task}!")

sequence sell() >>
    player_navigate(store_location)
    foreach(item in inventory.list_of_items) >>
        click_once(item)
        set_price(item)
    this.inventory.items = 0
    this.inventory.full = false

//-----
player MiningPlayer inherits GenericPlayer >>

:: MiningPlayer() >>
    this.task = "Mining"
    this.username = generate_username()
    inventory = Inventory()
    this.model = choose_ore_type()
    this.mining_location = "Varrock"
    this.store_location = "Varrock Bank"
    helper.player_instances++

async sequence engage() >>

    introduceSelf() => go_to_mining_spot()

    while(forever) >>

        await go_to_mining_spot()

        if(!click_model_once("${type}.png")) >> // Click node
            throw fish_spot_not_found

        if(inventory.check_if_full()) >>
            throw inventory_full

        item = await get_item() // Not implemented: Means whenever an item is collected

```

```

    if(!verify_ore(item)) >>
        throw wrong_item(item)
    else >>
        this.inventory.items++

    // Event Handlers
    catch(wrong_item(item)) >>
        drop_item(item)
            continue

    catch(inventory_full) >>
        sell()

    catch(fish_spot_not_found) >>
        go_to_mining_spot()
            continue

sequence go_to_mining_spot() >>
    player_navigate(mining_location)

sequence drop_item(item) >>
    right_click_once(item)
    click_drop(item)

function choose_ore_type() >>
    list = [gold, silver, mithril]
    type = list.select_one()

    return get_model("${ore_models}/{type}.png")

function verify_ore(model) >>

    if (model == this.model) >>
        return true
    else >>
        return false

//-----
player FishingPlayer inherits GenericPlayer >>

:: FishingPlayer()
    this.task = "Fishing"
    this.username = generate_username()
    inventory = Inventory()
    this.model = choose_fish_type()
    this.fishing_location = "Ardougne"
    this.store_location = "Ardougne Bank"
    helper.player_instances++

async sequence engage() >>

    introduceSelf() => go_to_fishing_spot()

    while(forever) >>

        await go_to_fishing_spot()

        if(!click_model_once("${type}.png")) >> // Click node
            throw ore_not_found

        if(inventory.check_if_full()) >>
            throw inventory_full

```

```

        item = await get_item() // Not implemented: Means whenever an item is collected

        if(!verify_fish(item)) >>
            throw wrong_item(item)
        else >>
            this.inventory.items++

        // Event Handlers

        catch(wrong_item(item)) >>
            drop_item(item)

        catch(inventory_full) >>
            sell()
            continue

        catch(ore_not_found) >>
            go_to_mining_spot()
            continue

sequence go_to_fishing_spot() >>
    player_navigate(fishing_location)

sequence drop_item(item) >>
    right_click_once(item)
    click_drop(item)

function choose_fish_type() >>
    list = [tuna, shark, mantaray]
    type = list.select_one()

    return get_model("${fish_models}/{type}.png")

function verify_fish(model) >>

    if (model == this.model) >>
        return true
    else >>
        return false
//-----

section Helpers >>

    // Helper that binds to all MiningPlayers, and FishingPlayers
    // Handles all static entities related to the players
    // 1:1 relationship between Player and Helper

helper PlayerVariables binds MiningPlayer, FishingPlayer >>

    // Static variables
    shared player_instances = 0
    shared net_mining_profit = 0
    shared net_fishing_profit = 0
    shared net_total_profit = 0

    // Generates a random username from a list of usernames present in a text file
function generate_username() >>

    list = get_local_file(usernames_path?global) // Explicit global reference (Scope
        resolution)
    rand = random_int(0,1000)

```

```

        username = list.as_list() => select_random()
        username = username.append(rand)

        return username
//-----
section Entities >>

class Inventory >>

    :: Inventory() >>
        this.items = 0
        this.slots = 28
        this.full = false

    function check_if_full() >>
        if(this.items == 28) >>
            this.full = true
            return true
        return false
//-----

```
