Logical Problem Solving

The goal of the document is to provide a refresh of information and to educate individuals on the importance of utilizing both critical thinking and all available tools when faced with programmatic problems.

Written by Michael Warmbier

Last Updated December 13th, 2022

Breakdown of Logic

Logic is a term we frequently use when describing solutions to problems. It refers to the validity proven by reasoning reliant on formal rules, whether defined by nature or the human mind. With computers, all logic resolves to simple *true* or *false* questions. The answers to these questions provide information to the machine that determines its actions. Programmers write *programs* that answer these questions.

This type of reasoning is purely theoretical and overly formal, but we can use it to make a thematically appropriate *logical conclusion* that if programs rely on providing gradual instructions, then we can represent those instructions linguistically as well.

Step By Step Programming

Breaking down a program into digestible tasks is an absolutely necessary skill in problemsolving. This may be experienced in several ways, including:

- translating code into language
- translating language into code
- translating complex language into simple language

Translating Code Into Language

Understanding the terminology used to describe concepts is important to be as concise as possible. Instructions such as:

```
int myValue = 3;
```

Example in C++

Can be described in a number of ways (in descending order of specificity):

```
- Create a variable with a value of 3
- Define an integer with the value of 3
- Declare an integer and assign it an initial value of 3
```

By being more specific in your phrasing, you may prevent more mistakes. As an example, the first description in this list does **not** specify the *data type*, which may be important in a *strongly typed* language such as C++ or TypeScript. Additionally, both the first and second descriptions in this list do **not** specify that the value is declared initially, which may or may not be relevant to the task.

Translating Language Into Code

The process of translating human language into code is often the easiest. Given the task:

```
- Define a function, isEven(), which takes an integer argument and returns
- Within isEven(), store the evaluation of whether the argument is even or
- Return the evaluation
```

This problem may be broken down into three steps, allow it to be easily translated into

pseudo-code:

```
function isEven(integer)
  boolean = (expression)
  return boolean
```

Pseudo-code is useful since it may be applied in the context of nearly any high-level programming language.

Note: just like with a typical exam problem, the expression to create the evaluation is up to you to determine. In the following examples, however, a general solution is provided:

```
int isEven(int operand) {          // Step 1
  bool eval = (operand % 2 == 0);  // Step 2
  return eval;                     // Step 3
}
```

Example in C++

```
def isEven(operand):               # Step 1
  eval = (operand % 2 == 0);       # Step 2
  return eval                      # Step 3
```

Example in Python

```
function isEven(operand) {        // Step 1
  let eval = (operand % 2 == 0);  // Step 2
  return eval;                    // Step 3
}
```

Example in JavaScriptM

Translating Complex Language Into Simple Language

It's common for questions given in college examinations and courses to be difficult to decipher and provided through unnecessarily large blurbs of text. You may be asked to solve a problem such as this:

```
"Functions are useful for completing simple tasks. For example, returning t
```

This is the same problem as above, but prior to being broken down into steps. In this state, it is much harder to keep track of any key concepts and words. As may be inferred, this small example may become much more problematic were it to be much larger and verbose:

```
"Create a UEmployee class that contains member variables for the university
```

Just like the previous example, this problem may be broken down into easy-to-read steps and notes:

```
/* Predfined Data */
- Create a class, UEmployee
  - Member Variable: Uni Employee
  - Member Variable: Salary
  - Member Method: Uni Employee GETTER
  - Member Method: Salary GETTER

- Create a class, Faculty, inherits UEmployee
  - Member Variable: Department Name
  - Member Method: Department Name GETTER/SETTER

- Create a class, Staff, inherits Uemployee
  - Member Variable: Job Title
  - Member Method: Job Title GETTER/SETTER

/* Program */
- Create instance of: UEmployee, Faculty, Staff
- Use all GETTER methods

/* Notes: */
* Setter methods must be defined, but are optional to use
* Variable names not specified
```

Although visibly taller, this version of the problem is much easier to isolate into sections and problems to solve one at a time. It makes use of notes to include information, either excluded or implied, as well as white space and indentation similar to something like Python.

Depending on what you use to write these steps down (either a specific application or through hand-written notes) you may even highlight certain keywords, such as **getter** and **setter** so that you are sure to remember them. It is always important to remember to keep track of any concept described in the problem in order to achieve the expected result, and not one similar though incomplete.

Troubleshooting

The developer environment (IDE) that you use for compilation can have a significant impact. This is true for many reasons but often overlooked when it comes to troubleshooting. *Errors*, whether thrown by the programmer manually or as a result of syntactical typos and such, may be described differently depending on the application.

The error `segmentation fault` is a common and easy-to-look-up error. This, combined with your language and possibly the description of your issue, make quickly troubleshooting the error as easy as a Google search. However, some compilers may return a less speific error, such as `memory error` or `stack overflow`. Even worse, you may come across a description as human unfriendly as the nothing but the addresses that the error occurred in, with no information as to what error even could be.

Given an error:

```
Assets\Scripts\ScriptBlock\myScript.cs(432,17): error CS1955: Non-invocable
```

This error comes from a Unity application and highlights the significance of memorizing terminology. Not every programmer uses Unity, but plenty of them understand the error provided that they can estimate a probable cause and possibly even offer a solution without that hands-on experience. Broken down:

- `Assets\Scripts\ScriptBlock\myScript.cs(432,17)` : The location of the issue in

location/file(line, character) format.

- `error CS19455` : The error code; likely already documented.
- `Non-invocable member: 'Transform.position' cannot be used like a method` : A description of the error, which suggests that a member is being misused as a method when it isn't one.

In this specific example, all of this information was provided to us. This may not always be the case. Potentially, we could get an error such as `core dumped` or `ERROR: #F32200, #F32201, #F32202, #C3B1302, #EFEC33` . These errors are, unfortunately, not very specific and don't leave us a lot to work with. If out of options, you may be inclined to share your problem with others who are more experienced. Doing this, however, will leave you with waiting time that will halt your progress.

Utilizing your IDEs built-in features is also fundamental. Features such as *break-points* and *debug mode* allow developers to trace problems and view the state of their program at specific points in runtime in order to narrow down the exact moment their issue occurs.

Documenting Code

Documenting key elements of your code is important for yourself and potential future contributors when creating larger projects. It does not rely on utilize standard naming conventions and relevant identifiers, however it is strongly encouraged to always title your data in an easy-to-understand manner. By documenting your code, another tool for diagnosing problems is available.

Code documenting, similar to translating complex language into simplified language in regards to problem solving, is entirely up to preference. It is, however, crucial to create documentation that is clear to the reader:

```
function identifier()
```

Description: *description*

Return Type: *type*

Arguments: *arg_name:arg_type(range of expected values) ...*

Handled Errors: *error title ...*

Documentation in this manner should also be done for globally used data, such as variables and objects. Official documentation of this type applied to a real project can be viewed in the developer documentation for Smite-API-Application or with a similar style on the official Game Maker Language documentation.

Note Taking Reference

Short-hand representations of logical concepts designed to resemble pseudo-code and make the translation from concept to implementation easier. This chart is intended to be a loose guide for the sake of understanding how to write good notes, not a catch-all reference associated with standard practice.

Programming Concept	Linguistic Description
Weak-Type Variables	VAR: `identifier`
Strong-Type Variables	VAR: `identifier` TYPE: `type`
For Loop	FOR: `iterator i ; condition a ; expression b`
While Loop	WHILE: `condition a`
Do-While Loop	DO-WHILE: `condition a`
Switch Statement	SWITCH: `condition a` CASE-VALUES: `range of case values` RESULT: `description of cases`
If/Else Statements	IF: `condition a` THEN: `relevant code` ELSE: `fallback`

Programming Concept	Linguistic Description
Else If	IF: `condition a` THEN: `relevant code` ELSE IF: `condition b` THEN: `relevant code`
Weak-Type Function	METHOD: `identifier()` ARGS: `argument1, argument2, ..` DESC: `description of function`
Strong-Type Function	METHOD: `identifier()` RET: `type` ARG(S): `argument1, argument2, ..` DESC: `description of function`
Classes/Objects	OBJ: `identifier`
Events	EVENT: `event type` DESC: `description of event`

Important Note: Signifying that an element belongs to a function locally or an objects list of members can be done in a plethora of ways, including a NOTE label, indentation and/or replacing VAR or METHOD with LOCAL-VAR, MEM-METHOD and MEM-VAR. This can also be extended to any optionally appendable attribute, such as constants, asynchronous behavior, ect.