**Montana Tech Novice Python Base Algorithmic Language Framework**Version 1.6  
December 2, 2015

A. Frank Ackerman

*Computer Science  
Montana Tech*

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 1.0 | 10/11/24 | Frank Ackerman | Initial version |
| 1.1 | 10/12/14 | Frank Ackerman | Add **Read** |
| 1.2 | 10/17/14 | Frank Ackerman | Drop terminating ; |
| 1.3 | 10/18/14 | Frank Ackerman | Read Obtain by **Prompt** followed by **Read** |
| 1.4 | 11/04/14 | Frank Ackerman |  |
| 1.5 | 11/22/14 | Frank Ackerman | Add **Else if** |
| 1.6 | 12/02/15 | Frank Ackerman | Change to Novice Python ALF |

**Montana Tech Computational Thinking Students:**

This standards encapsulate Dr. Ackerman’s decades of experience in the software industry, the IEEE software engineering standards, and many suggestions from various texts. You are a invited to participate in the continuing evolution of this by studying it critically and making suggestions for its improvement and correction.

Purpose

The purpose of this document is to define a base language for the expression of algorithms. This base language is intended to be sufficient for expressing algorithms that can be translated into simple Python computer programs. To this base language students may define additional constructs for algorithms that are not intended to be translated to computer programs.

Introduction

This document is intended to provides Computational Thinking students with an easy-to-use and easy-to-understand method for unambiguously describing algorithms that can be translated into computer programs and that can be unambiguously mentally “executed” by fellow students

The design elements defined in this standard are divided into the following categories:

### **3. Sequential Constructs**

**4. Selection Constructs**

**5. Repetition Constructs**

Overview

A Computational Thinking algorithm that can be easily translated into a computer program consists of a sequence of algorithm language constructs. Each construct begins with one of the keywords (initial letter capitalized and bolded) from the list below. Each element may begin with a label of the form D*dd* that is attached to the construct. When this is done all algorithm statements should be tabbed over so that the Ddd labels are all aligned the left margin.

As much as possible, language elements should be label in the sequence D00, D01, D02, .... . As the design develops and new elements need to be inserted, these labels may be extended by adding a suffix of a, b, c, ...

The design list of design constructs given in the next section is the heart of this document. To facilitate clear intent, and to make an algorithm abstractly executable, , only the listed constructs may be used expect where additional, problem specific constructs are defined.

The phrases after the keywords and optional comments are not explicitly defined but should clearly describe the intention or meaning of the construct for that algorithm, as should the optional comment following terminating (and labeled) right braces. Braces that may appear in a program should usually appear in the algorithm. Since algorithm text may be part of code, which almost always uses a single fixed width font, ordinary English words should not be used to name objects. Use compound names that clearly reference the object or attribute. For example, use wordCount instead of count to reference a word count.

# Sequential Constructs

|  |  |
| --- | --- |
| **Append** *listItem* **to** new*List* | Append *listItem* to the end of new*List* |
| **Call** *function [*with *parameters]* | to invoke a function |
| **Decrement** varName *by amount* | to describe that a value is being decremented (the amount is 1 if not stated)  This construction should always explicitly reference an object or attribute |
| **Increment** varName *by amount* | to describe that a value is being incremented (the amount is 1 if not stated.  This construction should always explicitly reference an object or attribute |
| **Obtain** *text* | Deprecated – use **Prompt** followed by **Read** |
| **Print** *text* | to display something on stdout |
| **Prompt** *text* | Display a prompt that the user needs to respond to. Should be followed by **Read**. |
| **Read** *[from file XXX] text***;** | To read data from a file. If a file is not referenced, stdin is assumed. |
| **Reset** *varName* to *text* | to change the value of an object or attribute back to something it was before |
| **Return** *[object values];* | to return from a function; |
| **Return** *to invoker* | to exit a program, return 0 for a normal exit and 1 for an error exit. |
| **Set** varName to *text* | to change the value of an object or attribute |

# Selection Constructs

|  |  |
| --- | --- |
| **Else {**  *body of else*  **}//Else** *optional comment* | the inverse of an If *condition* |
| **If** (*condition*) {  *body of if*  **}//If** *optional comment* | for code that is to be executed only if *condition* is true. |
| **If** (*condition*)  {  *body of if*  **}//If** *optional comment*  **Else if** (*condition*)  {  *body of elseif*  **}//Else if** *optional comment*  …  **Else**  **{**  *body of else*  **}//Else** *optional comment* | to describe a complex multi-way branch |

# Repetition Constructs

|  |  |
| --- | --- |
| **Break** | for code break |
| **Continue** | for code continue |
| **For** (*condition*) {  *body of for*  **}//For** *optional comment* | to describe loops that iterate across a sequence of items |
| **While** (*condition*) {  *body of loop* **}//While** *optional comment* | to describe a loop that will not be executed even once if *condition* is not meet  For a loop in which the exit condition is given by a break in the body of the loop, *condition*, is True. |

# Example

Problem statement: Any positive integer in [1, 2B] can be the start of an 3n+1 sequence. The sequence is constructed by halving the previous value if it is even or multiplying it by 3 and adding 1. Such a sequence is guaranteed to eventually generate the value 1. (If the starting number is 1, the sequence length is 1; if the starting number is 2, the sequence length is 2, as the sequence will be: 1, 1). An algorithm (using the constructs defined above) for obtaining a positive integer from a user (this is assumed) and computing the length of its 3n+1 sequence is given below. An example is:

Positive integer to start> 5

Length of this 3n+1 sequence is 6

The sequence in this example is 5, 16, 8, 4, 2, 1.

An algorithm that gives this result is:

A01 **Obtain** positive integer startInt from user:   
 “Positive integer to start> “

**Set** seqLngth to 1;

**Set** preSeqVlu to startInt;

**While** (True) {

**If** (prevSeqVlu equals 1) {

A02 **Print**: The length of this 3n+1 sequence is <seqLngth>;

**Break**;

}//If at end of sequence

**If** (prevSeqVlu is even) {

A03 Set nxtSeqVlu to prevSeqVul/2;

}//If

**Else** {

A04 Set nxtSeqVlu to (3\*prevSeqVlu) + 1;

}//Else

A05 **Increment** seqLngth;

**Set** prevSeqVlu to nxtSeqVlu;

}//While computing sequence values