Term Project Final Report

Dawn Programming Language

COIS-4100H

Sam Olan, 0674920

Oliver van Rossem, 0700291

Tobi Afolabi, 0625898

Table of Contents

Introduction

Dawn is a high-level Interpreted programming language designed to make programming accessible, especially for beginner programmers. Targeted primarily at learners with a grade 9 education level or higher, Dawn focuses on simplicity by using natural, English-like syntax and semantics. The language eliminates unnecessary complexities by including features such as global variables, 1-based array indexing, and intuitive operators, ensuring that new programmers can quickly grasp fundamental coding concepts. By blending ease of use with familiarity, Dawn provides an approachable starting point for learning programming while retaining concepts that align with more advanced languages, serving as a bridge for further exploration into the field of computer science.

Motivation

Dawn was designed to make programming easy for everyone, regardless of their prior knowledge. It isn't just another programming language; it's a new perspective on coding. Dawn's goal is to make coding easy and accessible for beginner's who are just learning to program.

Dawn is based off the simple idea that programming shouldn't be intimidating. We noticed that many existing languages were too complicated for beginners so we set out to create something different, something that would be easy for anyone to learn and use. What makes Dawn special is its simplicity. Unlike other languages that can be hard to understand, Dawn is straightforward. Its syntax is designed to be easy to read and write. Plus, Dawn encourages users to focus on understanding concepts rather than memorizing rules. This sets it apart from other languages and makes it a great choice for beginners.

Audience

While Dawn is suitable for people of all ages, its target audience is primarily young programmers between the ages of 12 and 18. This age group represents an important stage in education where curiosity and interest in technology often peak. By having a programming language that is tailored to their needs, Dawn aims to spark the passion for coding at an early age.

For beginner programmers, Dawn serves as a stepping stone towards learning more advanced concepts and languages in the future. While Dawn's target age is between 12 and 18, Dawn is also suitable for users of any age with education level grade 9 or greater, who are *dawning* their programming career. By laying a foundation in coding principles and practices, Dawn gives young programmers the skills and knowledge they need to tackle complex projects. Our language also encourages programmers to explore the fundamentals of coding in a supportive environment. Through learning resources, like code snippets from our IDE Dawn Online, beginners can build their skills and confidence as they are guided through the language. By providing a sense of curiosity and exploration, Dawn paves the way for lifelong learning and growth in the programming field.

Syntax and Semantics

The central principle behind the design of Dawn's syntax and semantics is ease of use. Designed for beginners with little to no programming experience, Dawn's syntax resembles the natural English language and enables a user to quickly express their ideas with minimal time spent learning form. For each operator in Dawn, there are special characters, such as +, and a corresponding English word, "plus" in this example. The user may express themselves in either way, whichever they are more comfortable with. Both readability and writability are supported by this principle. Should the user choose to use natural language, reading dawn code will not differ greatly from the experience of reading pseudo code. The syntax and semantics are such that in most cases reading a statement in Dawn will explain exactly what the line of code is doing. Below are key syntactic categories and how they are implemented in Dawn. For further information on the syntax of the categories mentioned, view the appendix.

Lexical Syntax:

Reserved Words: Selection statements

- if
- otherwise
- while
- for
- end

View Appendix A.1 for further information and code examples of selection statement syntax.

Reserved Words: Type specifiers

- int
    - integer
- float
    - Floating point
- String
    - String of characters. Abstraction of underlying structure, implemented as a one-dimensional character array
- list
    - Abstraction of underlying structure, implemented as a dynamic array.
- boolean
    - Boolean

View Appendix A.2 for further information and code examples of type specifier syntax and semantics.

Reserved Words: Arithmetic Operators

- =
- is
- +
- plus
- -
- minus
- *
- times
- /
- divided by

View Appendix A.2 for further information and code examples of arithmetic operators and their specific purposes, associated types, specifier syntax and semantics.

Reserved Words: Comparison Operators

- ==
- equals
- !=
- not equal
- ~=
- would be

View Appendix A.2 for further information and code examples of comparison operators and their specific purposes, associated types, specifier syntax and semantics.

Reserved Words: Other type-specific operators

- int:
    - absolute()
- float:
    - absolute()
- string:
    - [ ]
    - length()
- int:
    - length()
    - contains()
    - sort()
    - sortRev()

View Appendix A.2 for further information and code examples of other type specific operators and their specific purposes, associated types, specifier syntax and semantics.

Reserved Words: Other

- ( )
    - Used to execute a function on data member. The data member is placed inside the parenthesis
    - Used to enclose arithmetic expressions within an arithmetic statement
- “
    - Used to initialize the value of a string data member. The string value is placed inside the quotation marks
- &
- And
    - These are used to add short-circuiting functions to comparisons, enabling multiple comparisons in one statement. The comparison statement will evaluate to true if and only if all comparisons evaluate to true. If any evaluate as false, the entire statement evaluates as false. Further, if any evaluation as false, the remaining comparisons are not checked, as only one false evaluation is required to determine the entire statement is false. This is to the benefit of efficiency, as it is unnecessary to check statements following the false evaluation.
- |
- or
    - These are used to enable multiple comparisons in one comparison statement. If any comparisons in the statement separated by the operators are true, the entire

statement is true. If any comparisons are evaluated as true, the entire statement is true.

- ,
    - The comma is used to separate selection statement headers from their optional inline body statements
    - The comma is also used in sentences with multiple operations, to separate each operation
- ‘
    - The apostrophe is used after the name of a class object. It is used to specify that the following data member name is associated with the class object. Used to initialize class object data members.
    - BWM’s maxSpeed is 100
- has
    - used in class and object creation. Used to specify the attributes and associated functions in a class and or object creation.
- class
    - Used in class definitions. Designates the beginning of a class definition statement
- subclass
    - Used in subclass definitions. Designates the beginning of a subclass definition statement
- function
    - Used to designate the beginning of a function definition

In Dawn, the order of operators follows standard conventions found in mathematics. For example, multiplication and division take precedence over addition and subtraction, assignment operators take lowest precedence. However, to avoid ambiguity and ensure clarity, Dawn allows users to use parentheses to specify the order of operations explicitly.

Static Scoping

Static scoping determines how variable names are resolved based on the program's structure. This means that the scope of a variable is determined by its location in the source code and remains fixed throughout the program's execution. When a variable is referenced, the interpreter resolves its scope by looking at the structure of the program rather than its runtime behavior. Static scoping in Dawn ensures that variables declared within a block or function are accessible only within that block or function and any nested blocks or functions. This allows for clear and predictable variable resolution, as the scope of a variable is defined by its surrounding lexical context at the time of its declaration.

For example, if a variable x is declared within a function foo() in Dawn, it can only be accessed within foo() and any functions or blocks nested within foo(). On the other hand, variables declared outside of foo() are considered global variables and can be accessed from anywhere within the program.

By having static scoping rules, Dawn promotes code clarity and maintainability by minimizing unexpected variable interactions and ensuring that variables are used within their intended scope. This static scoping behavior provides developers with a clear understanding of variable visibility and facilitates the creation of modular and well-organized code structures.

Array Types

In Dawn, array types are designed to provide a flexible and efficient means of organizing and manipulating collections of data. When creating arrays, developers have several options available to define the size, type and initialization of arrays. This ensures that they meet the specific requirements of the program.

Grammatical Syntax

The grammatical syntax of Dawn is designed such that it resembles the natural English language. Similar to English, a *sentence* in Dawn is generally composed of two phrases. The first is the *noun phrase*, and the second is the *verb phrase*. The noun phrase is where the user addresses the nouns present in the sentence, the data members. In the verb phrase, the user specifies any actions to be performed on the data members, such as function calls, arithmetic operations, and comparisons. A sentence may have interior phrases, forming compound statements. These interior phrases can consist of noun and verb phrases. Connecting interior phrases are the conjunction operators. These consist of &, and, |, or. More information in Appendix A.4. Separating selection statement headers from their optional inline body statement is the comma. Additionally, the comma is used to separate operations in sentences with multiple operations.

Separating selection statement

View Appendix A.4 for further information and code examples of Dawn's Grammatical syntax.

Operator associativity rules:

- Arithmetic Operator Associativity
    - Left-to-Right
- Assignment Operator Associativity
    - Right-to-Left
- Comparison Operator Associativity
    - Left-to-Right
- Logical Operator Associativity
    - Left-to-Right

Classes and Class Objects:

Dawn includes the abstract data type, or in other words the *class,* and instances of classes*, objects*. This abstract data type, defined by the user, is used to represent more complicated data structures, in a readable and understandable way. They may contain multiple data members, known as attributes. These abstract data types, or objects, may have associated operations, functions, defined by the user. A class may have many instances of itself, or in other words, there may be many objects of the same class, the same abstract data type. Any object of a class must have the same number, and the same type of attributes as the parent class. Dawn supports child classes. Child classes inherit class attributes and methods from their parent classes. These methods may be overwritten in the child classes.

The inclusion of classes and class objects in Dawn is paramount to the principle of user-friendly experiences. This is due to the concept of encapsulation. Abstract data types, classes, and instances of classes, objects, encapsulate data and processes from the user. This supports readability, writability, and understandability. Dawn predefines a library of classes and class functions, allowing the user to interact with classes without being concerned with implementation. This abstracts processes from the user, allowing them to express without requiring a deep understanding of class and object definition.

Interaction with the object-oriented paradigm in Dawn follows the language's driving principle, ease of use. A programmer may define classes and create instances of classes using natural English language. Similar to other functions and data members, the user may use natural language to interact with the classes and objects, as well as their attributes and functions.

View Appendix A.5 for further information and code examples of class and object creation syntax and semantics.

The decision to support classes and objects in Dawn is primarily because of their ability to encapsulate data and processes, hiding them from the user. This promotes readability and understandability, especially for beginner users. It also acts as a good introduction to the powerful tool of object-oriented programming, further increasing Dawn's appeal. The decision to make an object extend the same number of attributes as its parent class(s) was made to improve language orthogonality. In the initial development of Dawn, an object could have a dynamic number of attributes. This was to allow the user to freely create attributes as they saw fit. However, it creates the following issue: the user creates a class car, with attribute speed. Then, they create an object of type car called BWM, speed equal to 100. They also then realize that they want another attribute for their car, so they give BWM another attribute, colour, and initialize it to blue. This won't break the user's program, but it will allow them to build a bad programming habit. Moving forward, the user will have to define a new attribute colour for every new instance of car. Instead, it is better to encourage the user to set the necessary attributes in the class definition, not in the object definitions. A user is not however required to initialize all attributes of a class upon object creation.

This allows the user to initialize the values later, should they not be ready to when designing and creating their objects. A user may also declare class and object functions, without immediately defining them. Dawn also supports object inheritance, where an object inherits the attributes and operations of their parent class(s). Multiple inheritance is supported. *View Appendix A.5 for further information and code examples of class and object inheritance syntax and semantics*.

Dynamic binding

Dawn supports dynamic binding. This is to allow for polymorphism, supporting the object-oriented paradigm. Upon runtime, the interpreter binds values to attributes, and operations to classes. This ensures that any overridden class or object functions are bound to the correct class or object.

Functions:

To signify the start of a function definition, Dawn uses a function definition header starting with the keyword function, followed by the function name, and opening and closing parentheses. Dawn only uses global variables which simplifies the language's design and usage. With only global variables, there's no need to manage parameters or worry about scoping within functions. All variables are accessible from anywhere in the code, making it straightforward to work with data across different parts of the program. Below the function header is the function body. Rather than using opening and closing parentheses or braces to enclose functions, Dawn uses the function definition header combined with a function closing statement. As a result, the function body is not required to be indented. The syntax of this statement is the keyword end, followed by the function name. View Appendix A.3 for example.

Indexing

It’s common for an array index to begin at 0 in programming languages, however in Dawn, an array index begins at 1. This decision was made to improve readability for beginner users. Outside of computer science, it is common practice to consider the first item in a collection of items as the first item, not the zeroth item. So, users may be confused to see that they interact with the second item in a collection when addressing index 1, were the index to begin at 0.

Variable Lifetime

In Dawn, variables have static lifetime which persist throughout the entire duration of the program's execution. This retains their values from the moment they are initialized until the program terminates. This concept applies to global variables which are declared outside of any function or block. They are allocated memory when the program starts and maintain their values across all functions and blocks within the program. Additionally, constants declared using the const keyword also have static lifetime, maintaining a constant value throughout the program. By having static lifetime for variables, Dawn ensures the consistency and accessibility of certain data across different parts of the program, facilitating the management of program state and aiding in the development of predictable software solutions.

Online IDE (Dawn Online)

Accompanying the launch of the Dawn programming language is Dawn Online, an intuitive, browser-based Integrated Development Environment (IDE) designed to streamline the process of learning and coding in Dawn. With Dawn Online, users can write, test, and debug Dawn code without the need to install any additional software. This accessibility makes it ideal for beginners and seasoned programmers alike, as it removes barriers to entry and allows for quick experimentation with the language.

One of the key features of Dawn Online is its use of pre-defined code templates for common programming uses. These templates are used for a wide range of scenarios, including the creation of base and inheritance class structures, method class structures, and if-else statements. By taking advantage of these templates, users can improve their coding process and ensure consistency in their code structure.

Dawn Online also offers integration with cloud storage services, allowing users to save their projects and code snippets directly to the cloud. Not only does this ensure that their work is safely backed up but also allows collaboration with other programmers or access to their code from multiple devices. The cloud storage feature enhances the portability and accessibility of Dawn projects which allows the users to work on their code from anywhere with an internet connection.

Orthogonality

What is Orthogonality?

Orthogonality is an important subject as it defines a fundamental principle in the independence between language features. It refers to how a relatively small set of primitive constructs can be combined in a small number of ways to build control and data structures. Also, every possible combination is legal with few exceptions to the rules. This characteristic is pivotal for Dawn as it significantly impacts the flexibility and simplicity of a language.

A language exhibiting high orthogonality allows for different features to be combined in various ways without imposing restrictions. This means that programmers have the freedom to express their solutions in different manners which leads to code that is not only concise but also more expressive and maintainable. Imagine a language where you can seamlessly combine control structures like loops and conditionals with data structures like arrays and lists, without having to worry about arbitrary limitations or inconsistencies. This level of consistency and freedom is what Dawn will bring for beginner developers. However, achieving high orthogonality requires careful design and devotion to certain principles. Regularity, for instance, plays an important role in improving orthogonality. By ensuring that language constructs follow consistent patterns and rules, developers can better understand how different features interact and combine, reducing the cognitive burden associated with learning and using the language.

Also, orthogonality is not just a matter of convenience; it directly impacts the scalability and maintainability of software systems. Languages with low orthogonality often require developers to memorize numerous special cases and exceptions, leading to code that is harder to comprehend and modify over time. On the other hand, languages with high orthogonality allow developers to write code that is not only more elegant but also easier to extend and refactor as project requirements evolve. Orthogonality serves as a cornerstone in the design of programming languages which enables the construction of robust, expressive, and maintainable software systems by promoting consistency, flexibility, and simplicity in language features and constructs.

Consistency in Syntax and Semantics

Consistency in syntax and semantics is the foundation for effective language design, offering programmers a framework for code implementation. This section describes the importance of unified syntax, predictable semantics, and the reduction of special cases in Dawn.

Unified Syntax

In the context of Dawn, a unified syntax is the core principles of consistency and simplicity across its various language constructs. Named "Dawn Syntax," this approach ensures that programmers encounter familiar patterns and conventions throughout their code, regardless of the specific task at hand. This plays a vital role in simplifying the learning curve for beginners entering the Dawn ecosystem. With Dawn Syntax, newcomers are presented with a framework that reduces the need for memorizing difficult syntax rules. Instead of handling an array of syntax details, learners can quickly understand the principles and patterns of Dawn. This learning curve encourages beginners to progress quicker from basic syntax understanding to skill in writing meaningful Dawn code.

Dawn Syntax basically acts as a mechanism for programmer productivity within the Dawn language ecosystem. By relieving developers from difficult syntax rules, Dawn allows beginner programmers to focus on their ability to understand the meaning of the code they're writing. This difference in focus from syntax to problem-solving not only improves the development process but also promotes a better understanding of programming concepts and best practices within the Dawn community. Thus, the adoption of Dawn Syntax represents a fundamental step towards creating a language ecosystem that unleashes the creative potential of developers worldwide.

Predictable Semantics

Predictable semantics ensure that language constructs behave intuitively and consistently, aligning with developers' expectations. A natural programming language allows developers to anticipate code execution based on their understanding of common programming concepts. This predictability builds confidence in code correctness and helps debugging and maintenance efforts.

Elimination of Special Cases

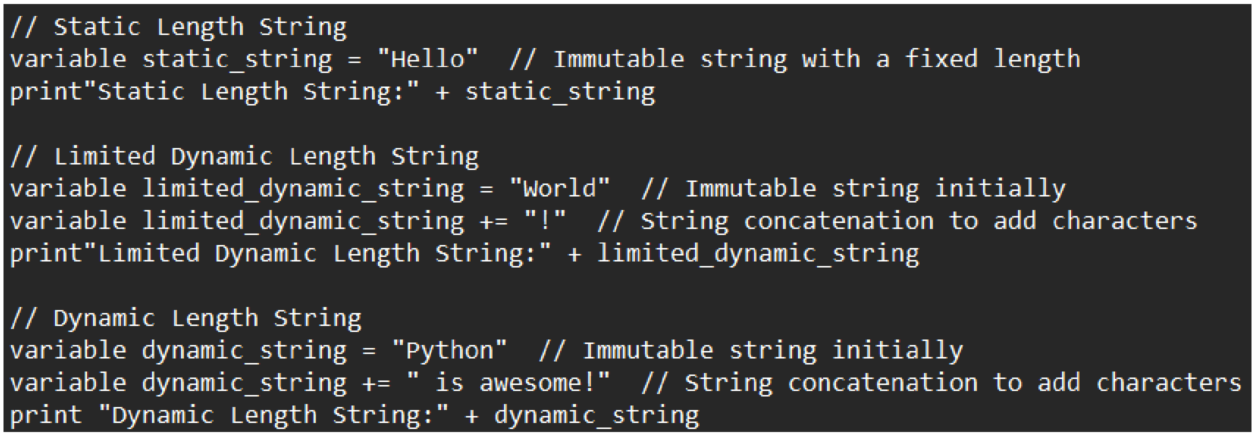
Minimizing special cases or exceptions to the rules is another importance of language consistency. By observing consistent design principles, language designers can lower limitations or irregularities that delay how the language is expressed. A reduction in special cases enables developers to rely on a more uniform and intuitive set of language features, resulting in concise, readable, and maintainable code.

Uniformity in Data Types

What is Uniformity?

Uniformity in data types lies at the core of programming language design and it defines a consistent framework for representing values and performing operations on them. In Dawn, data types have a range of categories, from primitive types to complex structures, each characterized by predefined operations and attributes. Primitive data types in Dawn include integers, floating-point numbers, booleans, characters, and strings, each with specific attributes and operations tailored to their respective needs. For instance, integers have variations like byte, short, int, and long, accommodating different ranges and precision levels. Similarly, floating-point numbers encompass float and double types, representing real numbers with varying degrees of precision. These data types do not need to be specified during declaration since the Dawn is dynamically typed.

Strings, a fundamental data type in many programming languages, are treated as a special kind of character array in Dawn. However, in Dawn, strings are exclusively immutable. They come with options for static, limited dynamic, or dynamic lengths. The representation of strings in Dawn ensures efficient manipulation and storage, catering to diverse application requirements.



```
// Static Length String
variable static_string = "Hello"  // Immutable string with a fixed length
print"Static Length String:" + static_string

// Limited Dynamic Length String
variable limited_dynamic_string = "World"  // Immutable string initially
variable limited_dynamic_string += "!"  // String concatenation to add characters
print"Limited Dynamic Length String:" + limited_dynamic_string

// Dynamic Length String
variable dynamic_string = "Python"  // Immutable string initially
variable dynamic_string += " is awesome!"  // String concatenation to add characters
print "Dynamic Length String:" + dynamic_string
```

Descriptors play a crucial role in defining and managing data types in Dawn. They encapsulate the attributes associated with variables, providing a means to access and manipulate data in a structured manner. Descriptors facilitate the organization and manipulation of data elements, ensuring coherence and consistency in the language's data model.

Dawn supports complex data structures such as enumerations, arrays, associative arrays, records, tuples, and union types, each offering unique capabilities for organizing and accessing data. Enumerations provide a predefined set of named constants, while arrays enable the creation of homogeneous aggregates of data elements indexed by position.

Furthermore, Dawn incorporates advanced features like associative arrays, which allow for unordered collections of data elements indexed by keys, and record types, which aggregate data elements identified by names. These features enhance the expressiveness and flexibility of the language, empowering developers to model complex data relationships effectively.

Interpreted Language

Dawn is an interpreted language. Since Dawn is designed for beginner programmers who are learning to program, it is crucial that they be provided with quick feedback on changes to their code. So, rather than waiting for the intermediate step of compilation, Dawn is interpreted at runtime. This allows the programmer to immediately execute their code after any changes, allowing for faster testing and debugging.

Since Dawn is likely to be used by beginners who are learning to program, it is unlikely that any lost execution speed from lack of compilation will be noticed, thus making the potential for slower execution speed worth the trade-off for faster feedback. Additionally, the decision to make Dawn an interpreted language rather than compiled lends itself to better portability. The interpreter programs required to execute Dawn code can be ported to different systems (Mac, windows, etc.) avoiding the need for any changes to user source code or the creation of compiled executable files specific to each system. This means that so long as the system supports the interpreter (interpreting program), Dawn's source code can be executed on any platform easily. This also enables Dawn Online, since there is no need for the creation of executable files.

Dynamically Typed

Dawn is a dynamically typed language. Given that beginner users are likely unfamiliar with the concept of types, and the language is designed to allow beginners to focus on design before implementation, Dawn allows users to use data members without declaring their type. The type of any created data member is inferred dynamically by the interpreter at runtime. The user can explicitly state the type of the variable if they wish, but they are not required to do so. This enables user learning, as they are not limited by their understanding of types, but should they be interested, can declare types themselves.

Types in Dawn

Dawn is dynamically typed, so the user is not required to specify data member types. However, the user is enabled to specify data member types should they so desire. This allows a user the opportunity to progress their learning, experimenting with types themselves. The types handled in Dawn are as follows:

- integer
- floating point
- string
- list
- Boolean

Note that there is no char type to represent a single character. In Dawn, a single character is treated as a string of length 1.

For more information on types in Dawn, view section A.2 in the appendix. Here, the types and their related operators are detailed, as well as the associated reserved keywords.

Concurrency

Concurrency is the ability of a program to execute multiple tasks simultaneously, and it can perform at different levels of the system. In Dawn, concurrency is essential for scalability and portability which brings efficient execution of tasks across various architectures.

Levels of Concurrency

Concurrency can occur at multiple levels in Dawn:

- Machine Instruction Level: Executing multiple machine instructions concurrently.
- High-Level Language Statement Level: Executing multiple statements of a high-level programming language concurrently.
- Unit (Subprogram) Level: Executing multiple subprograms or tasks concurrently.
- Program Level: Executing multiple programs or processes concurrently.

Multiprocessor Architectures

Dawn can control different multiprocessor architectures, such as Single-Instruction, Multiple-Data (SIMD) machines and Multiple-Instruction, Multiple-Data (MIMD) machines. These architectures provide the hardware foundation for executing concurrent tasks efficiently.

Categories of Concurrency

Dawn supports two main categories of concurrency:

- Physical Concurrency: Involves multiple independent processors executing tasks concurrently.
- Logical Concurrency: Mimics physical concurrency by time-sharing a single processor which allows software to be designed as if multiple tasks are executing simultaneously.

Motivations for Concurrency

Concurrent execution in Dawn offers several benefits:

- Speed of Execution: Multiprocessor systems can execute tasks faster by parallelizing computations.
- Efficient Resource Utilization: Even on single-processor systems, concurrent programs can be more efficient by overlapping computation with I/O operations.
- Scalability and Portability: Concurrent programming enables scalable and portable software solutions, suitable for distributed systems and diverse hardware architectures.

Subprogram-Level Concurrency

Dawn facilitates subprogram-level concurrency through tasks or processes. Tasks are program units capable of concurrent execution and communicate with each other using mechanisms such as nonlocal variables, message passing, or parameters.

Synchronization

Concurrency in Dawn requires synchronization mechanisms to coordinate access to shared resources:

- Cooperation Synchronization: Ensures that tasks wait for each other to complete specific activities before proceeding.
- Competition Synchronization: Prevents conflicts when multiple tasks access shared resources simultaneously, typically achieved through mutually exclusive access.

Design Issues for Concurrency

Developers using Dawn must consider various design issues related to concurrency, including synchronization, task creation and execution, and choice of synchronization mechanisms. Dawn provides support for synchronization through semaphores, monitors, and message passing, allowing developers to design efficient concurrent programs.

Conclusion

Dawn represents a groundbreaking approach to programming language design which is driven by the motivation to make coding accessible and intuitive for beginners. By prioritizing simplicity and readability, Dawn stands out form other object-oriented languages and offers a natural language syntax and semantics. Its focus on design-first thinking encourages users to understand concept instead of memorizing rules. Dawn aims to give a passion for coding at an early age and serving as a stepping stone for learning more advanced concepts and languages in the future. Through the IDE Dawn Online and other learning resources, Dawn encourages beginners to build their coding skills with confidence. With its orthogonality, uniformity in data types, and innovative features such as natural language input, Dawn is a step forward in programming language design. Its interpreted nature and dynamic typing improves its accessibility and portability making it suitable for a wide range of platforms.

Appendix

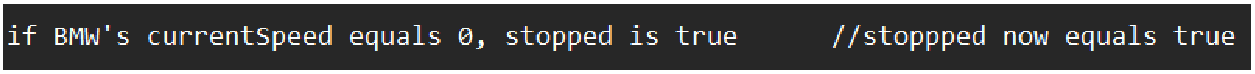
A.1: Selection Statement Syntax

If statement:

if *comparison*, *optional inline statement*

```
        body
otherwise, optional inline statement
        body
end if
```

If statements are used in Dawn to control the flow of a program based on certain conditions. The syntax for an if statement in Dawn is straightforward. The ‘if’ keyword is followed by the condition to evaluate. If the condition is true, the code block following the ‘,’ is executed. Optionally, an else block can be provided to specify code to execute if the condition is false.



```
if BMW's currentSpeed equals 0, stopped is true      //stoppped now equals true
```

Loops:

While loop:

```
while comparison, optional inline statement
        body
 end while
```

Here, the optional inline statements, designated by the comma following a comparison in the header, are only applicable to selection statements with no further body. These allow the user to have a selection statement and single-line body in one line, with natural language syntax.

For loop:

```
For member in collection
        body
end for
```

A.2: Lexical Syntax and Examples

Data members are listed in the tables below, along with their associated operators, their natural language equivalent (if applicable) and a brief description of each operations function. In some cases, x will represent a data member. Examples of unique operations follow the tables.

int	operation	purpose
Arithmetic:		
	+ or “plus”	Addition
	- or “minus”	Subtraction
	* or “times”	Multiplication
	/ or “divided by”	Division
Comparison:		
	== or “ equals ”	Equal to
	!= or “ not equal ”	Not equal to
	< or “ under ”	Greater than
	> or “ over ”	Less than
	<= or “is or over ”	Less than or equal
	> or “is or under ”	Greater than or equal to
See example	~ or “ would be ”	Comparison after arithmetic operation without modification
Other:	absolute()	Returns absolute value of int

float	operation	purpose
Arithmetic:	Same as int	Same as int

Comparison:	Same as int	Same as int

string	operation	purpose
Arithmetic:		
	+ or “plus”	Concatenates
See example	/ or “contains”	Checks for occurrence of a substring
Other		
See example	[*x*]	Access individual character at index x in string
	length()	Returns the length of string

list	operation	purpose
Arithmetic:		
See example	Same operators as int	Attempts operation on specified member of list if same type or through narrowing conversion
Comparison:		
See example	Same as int	Attempts comparison of specified member of list if same type or through narrowing conversion
Other:		
	length()	Returns the number of elements in the list
	contains()	Checks if element occurs in the list

See example	[ ]	Checks member at specified index
	sort()	Sorts the list in ascending order
	sortRev()	Sorts the list in descending order

<u>boolean</u> <u>Comparison</u>:	operation	purpose
	== or “ equals ”	Equal to
	!= or “ not equal ”	Not equal to

List item initialization

A user can initialize an item by specifying the desired index using [ ], followed by the value to be inserted. See List operators below for further details.

Examples of unique operators:

~ , would be

if x plus y would be under or equal z, x plus y
if x + y ~ <= z, x + y

This operator, equivalent to the “would be” operator in the language, is used with a comparison operator. Together, the ~ and comparison operator compare the result of some arithmetic expressions against some value. Despite the comparison taking place after the arithmetic expression, the ~ operator determines that the arithmetic expression is executed on a copy(s) of the data member(s) on the left-hand side of the statement, not modifying the values of any data members involved in the arithmetic expression. The ~ operator must proceed the comparison operator.

In these equivalent cases, the arithmetic expression x + y is not carried out on the actual parameters. It's performed on copies of their values, and the result is compared against the right hand side of the statement (right hand of comparison operator)

String

```
string x, y
 x / y
```

Since these operators follow the language's associativity rules, the arithmetic expression x / y is evaluated from left to right. So, the statement checks to see if the string x contains any instances of the string y and returns the number of occurrences. So, if there were two instances of substring y in x, the statement would return 2.

String arithmetic [ ]

```
string x is “contain”
boolean contains

if x[4] == “t”, boolean = true
Otherwise, boolean = false

if x[4] == “tains”, boolean = true
Otherwise, boolean = false
```

In this example, [ ] is used to check if the member at the specified index. On the fourth line, the string x which is initialized with “contain” is checked at index 4 for the character t. On the seventh line, the fourth position is checked for the substring “tains”. In a scenario such as this where the [ ] operator is compared against a string rather than a character, the interpreter checks for an instance of the substring, starting at the index specified. So, in this case it is looking for the substring “tains”, starting at index 4, and is evaluated to be true.

List [ ]

```
list x
if x[1] equals 10
        body
end if

list x
…
x[10] is “10th item in list”
```

We can see in these examples that the square brackets allows the user to access an element of a list at a given index

List Arithmetic Operators

```
list x
list[2] + 5
list[3] + “string”
list[4] / 10
```

In these examples, we see how arithmetic operations may be performed on list elements. In the second line, the interpreter checks if the element at list index 2 exists. If it does not, it treats it as a null integer equivalent to 0, and adds 5 to it. If the element does exist, it checks if the type is compatible with the type of the list. If it is, it performs the appropriate operation, plus 5 in this case. If the types are different, the type will not be coerced, as this could lead to complications. In the second line, the interpreter checks if the third element exists. If it does not, it creates it and initializes it to the operand on the right side of the operator. If it does exists, and the list type string, the string on the right side of the operator is concatenated onto the end of the string. If it is not a string, an error is generated.

List sort()

```
list x = [4, 2, 9, 0]
sort(x)

list y = [“banana”, “apple”]
sort(y)
```

In the first example, the result of the sort function executed on the list x is x == [0, 2, 4, 9]. In the second example, we see the sort function on a list of type string. In this case, the sort lists the strings in ascending order based on the first character in each string. The result is y == [“apple”, “banana”]

A.3: Function Declarations Syntax

```
function functionName
        function body
end functionName
```

A.4: Grammatical Syntax

```
string x is “string”
```

In this example of a sentence in Dawn, we see a syntax similar to that of English, composed of the noun phrase and the verb phrase. Highlighted in yellow is the noun phrase, where the user specifies relevant nouns, i.e relevant data members. Highlighted in blue is the verb phrase, where the user specifies the verbs, i.e the actions performed on the relevant data members.

```
if x plus y equals z and z under 10, a is 10
```

In this compound statement, the syntactical rules are still followed. We can see that the interior statements, highlighted in yellow, position any nouns on the left side of the interior statement, and the verbs on the right. Nouns may appear to the right of a verb that is already applied to a noun.
Note:
“y equals z” is an interior verb phrase.
“and” is a conjugation, connecting interior phrases, similar to English.

A.5: Class and Object Syntax

```
class car has colour, maxSpeed, function drive, function park

car BMW’s colour is blue, maxSpeed is 100


function BMW’s drive()

        function body

end BMW’s drive


BMW’s subclass sedan has colour red

function sedan’s drive()

        function body

end sedan’s drive
```

In this example, we see the creation of a class, car in this example. The creation resembles natural English language and can be done in one line without initialization. Notice that the class functions can be declared and not defined, supporting design before implementation. The has keyword is used to bind attributes and functions to a class. The ‘s keyword is used to initialize object attributes associated with a class and to define associated functions. Notice that any object has the same number of attributes as its parent, or less (an attribute need not be initialized).

We also see how inheritance is defined. To define a new object as a subclass of another object, the user uses the ‘s keyword. The ‘s is applied to the name of the parent, followed by the keyword subclass, and then the name of the new object. Following are any attribute initializations or new function declarations. We also see an example of polymorphism. A user may override an objects parent functions by specifying a new function with the function keyword, followed by the objects name with the ‘s keyword, and the name of the function to be overridden. If the function name is not an existing function in the parent, a new function is created for the child. If the function name does match that of a parent function, then the function is overridden.

```
class car has colour, function drive, function park, maxSpeed
```

Notice that this line differs in attribute order from the previous class definition. In Dawn, the order of class or object attributes, as well as function declarations, do not impact the creation of said structures.