Algonquin College Logo

# SCHOOL OF ADVANCED TECHNOLOGY

### ICT - Applications & Programming

### Computer Engineering Technology – Computing Science



A11

Language Specification

Lab Professor / Lab Session:

Paolo Sousa / 012

Team:

Tasfia Khan Aohana - ID: 041076039 / Philogene Villanueva - ID: 041064813

Language Name: Amber

|  |  |
| --- | --- |
| **Part**  **1** | **Language User Reference** |

**EXPLANATION**

The purpose of this assignment is to invent a new computer language.

* This language can have the syntax and structure of your choosing.
* Option 1: Adapt the ‘Sofia language to be Go compatible (see <https://go.dev/>).

This is going to be a basic language. There's a lot of functionality that we'll be skipping over, while we implement the basics. You will need to tell me those basics, of course. In this document, I'm going to explain the steps of what to do with a bit of detail.

* 1. **User Manual**

**Element 1: Name / Extension**

Name of Language: **Amber**

Extension: **.amb**

The Language is inspired by: **C, Java, Go**

**Element 2 – Comments**

**Single line comment syntax: //**

*Example :*

//This is a single-line comment.

**Multi-line comment syntax: /\* \*/**

*Example :*

/\*

This is a multi-line comment.

It can span multiple lines and is enclosed

between /\* and \*/.

\*/

**Element 3 – Keywords**

break default func interface select  
case defer go map struct  
chan else goto package switch  
const fallthrough if range type  
continue for import return var

**Element 4 – Datatypes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Keyword** | **Number of Bytes** | **Range/ Description** |
| Integer | int32 | 4 | Signed 32-bit integer, range from -2,147,483,648 to 2,147,483,647 |
| Boolean | bool | 1 | Represents true or false values |
| Float | float32 | 32 | Single-precision floating-point number, roughly 7 decimal digits of precision |
| String | string | variable | A sequence of bytes representing a text string of variable length |
| Char | char | 1 | A single byte representing a character or small integer value, typically ASCII characters or extended character sets. |

**Element 5 – Variables**

Syntax for variable decalaration:

var variable\_name data\_type

*Example:*

var a bool // Declaring variable ‘a’ of Boolean data type

var i, j, k int32 // Declaring multiple variables of type int32

var b, salary float32

var d string = “Hello World!”

**Element 6 – Methods / Functions**

**Functions** are defined using the **func** keyword followed by the function name, parameters, return types, and the function body.

func function\_name( [parameter list] ) [return\_types] {

//body of the function

return return\_value

}

*Example: A function that receives two integers as parameters and returns their sum as an integer.*

func myFunction(x int, y int) int {

return x + y

}

*Example 2: A function that returns an integer and a string message.*

func myFunction2(x int, y int, username string) (sum int, message string) {

sum = x + y

txt = “Hello ” + username

return

}

**Methods**, are functions that are associated with a specific type. They operate on instances of that type and can access and modify the data within those instances. Methods are defined similar to functions but with a receiver parameter, which specifies the type to which the method is attached.

func (receiver parameter) method\_name( [parameter list] ) [return\_types] {

//body of the method

return return\_value

}

*Example: This is a method that receives a Circle struct, and calculates the area of a circle, given the radius defined in the instance of the struct.*

func (c \*Circle) CalculateArea() (área float32){

area = math.Pi \* c.Radius \* c.Radius

return area

}

**Element 7 - Commands**

**Assignment:** To assign a value to variable, the assignment operator (=) is used. The syntax is as follows:

// Assigning value to an existing variable

variableName = value

// Declaring and initializing a new variable

variableName := value

**Type conversion:**

Type conversion is allowed given that it’s safe and meaningful, with no unintended data corruption. To perform type conversions, the following syntax is used:

newTypeValue := targetType(expression)

// Example – converting int32 to float32, and storing data in new variable ‘b’

var a int32 = 7

b := float32(a)

**Math Operations:**

Standard arithmetic operations can be performed using the respective mathematical operator, like addition (**+**), subtraction (**-**), multiplication (**\***), division (**/**), and modulus (**%**) for integers and floating-point numbers.

In addition to the standard operations, more advanced mathematical functions can be implemented using the ‘**math’** package in the standard library. For example, exponentiation and trigonometric functions. To import the math package, include the following line of code:

import "math"

// Example

sqrt := math.Sqrt(25) // Square root

sine = math.Sin(math.Pi) // Sine

exp = math.Pow(2, 3) // Exponentiation (2^3)

**String Concatenation:** strings can be concatenated using the + operator.

// Example

str1 := "Hello "

str2 := "World!"

message := str1 + str2 // message will contain the text "Hello, World!"

**Selection structures:** To conditionally execute blocks of code based on the evaluation of boolean expressions, the following conditional statements are used: if, if-else, switch and select.

* **Boolean Operators:**

|  |  |
| --- | --- |
| **Operation** | **Operator** |
| Logical AND | && |
| Logical OR | || |
| Logical NOT | ! |
| Equal to | == |
| Not Equal to | != |
| Less than | < |
| Less than or equal to | <= |
| Greater than | > |
| Greater than or equal to | >= |

**If-statement:**

if condition {

// Code to be executed if the condition is true

}

**If-else statement:**

if condition {

// Code to be executed if the condition is true

} else {

// Code to be executed if the condition is false

}

**If-else if-else statement*:***

if condition1 {

// Code to be executed if condition1 is true

} else if condition2 {

// Code to be executed if condition2 is true

} else {

// Code to be executed if neither condition1 nor condition2 is true

}

**Nested if statement:**

if condition1 {

// Code to be executed if condition1 is true

if condition2 {

// Code to be executed if both condition1 and condition2 are true

}

}

**Switch statement:** In a switch statement, each case contains expressions, which is compared against the value of the switch expression.

switch expression {

case value1:

// Code to be executed if expression == value1

case value2:

// Code to be executed if expression == value2

// ...

default: // This is an optional default case

// Code to be executed if no cases match

}

**Select statement:** used for communication and synchronization between routines when working with channels. It provides a way to choose which of several channel operations to proceed with, depending on which operation can proceed immediately. A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready.

select {

case channelOperation1:

// Code to be executed if channelOperation1 can proceed

case channelOperation2:

// Code to be executed if channelOperation2 can proceed

// ...

default:

// Code to be executed if no channel operation can proceed immediately

}

**Iteration Structures:** The following looping structures are used: for-loop, continue, break, range and nested for-loops.

1. **for-loop**

for statement1; statement2; statement3 {

// Code to be executed for each iteration

}

* + ***statement1*** - Initializes the loop counter value.
  + ***statement2*** - Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop exits.
  + ***statement3*** - Increases the loop counter value.

1. **continue** **statement** - is used to skip one or more iterations in the loop. It then continues with the next iteration in the loop.
2. **break statement** -  is used to break/terminate the loop execution.
3. **Nested for-loops**

// Outer loop

for statement1; statement2; statement3 {

// Code to be executed for each iteration

for statement4; statement5; statement6 { // Inner loop

// Code to be executed for each iteration

}

}

**Input:** All inputs are taken in as Strings.

* + To get input from the keyboard (stdin), the fmt package's Scan, Scanf or Scanln functions are used to read data as strings from the standard input.
    - **Scan()** - receives the user input in raw format as space-separated values and stores them in the variables. Newlines count as spaces.
    - **Scanln**() - similar to Scan(), but it stops scanning for inputs at a newline (at the press of the Enter key.).
    - **Scanf()** - receives the inputs and stores them based on the determined formats for its arguments.

**Output:**

* The fmt package's Print, Printf, or Println functions are used to display values stored in variables or a string of characters.
  + **Print()** - prints its arguments with their default format.
  + **Println()** - similar to Print() with the difference that a whitespace is added between the arguments, and a newline is added at the end
  + **Printf()** - formats its argument based on the given formatting verb and then prints them.
    - %v is used to print the value of the arguments
    - %T is used to print the type of the arguments

**Functions:** defined using the func keyword. Functions can take parameters, and they can also return values.

* *Syntax for function declaration:*

func function\_name( [parameter list] ) [return\_types] {

//body of the function

return return\_value

}

**Element 7 – Proper elements**

* **Enhanced Documentation tools**, including special comment syntax for documentation purposes. The documentation comments can be defined as:

*/\*\**

*\* This is a function called functionName, which …*

*/*

* **Incorporate more data types**. such as float64, rune and complex64, which enables us to handle larger data types and deal with complex numbers and Unicode characters.
* **Generics -** to write functions, classes, and data structures that can work with a wide range of data types while maintaining type safety. This will allow for code to be reused and increase flexibility and enhance performance.
* **Error Handling and Debugging –** Introduce more concise ways to handle errors and debug code, with more descriptive error/warning messages that allow for a more comprehensive understanding of errors encountered.

|  |  |
| --- | --- |
| **Part**  **2** | **Language Comparison** |

**Comparing with C language**

**Differences**

|  |  |  |
| --- | --- | --- |
|  | * **Syntax**: Every programming statement in C must be followed by a semicolon. This informs the compiler of the boundary between one statement and the subsequent one. Semicolons are acceptable in Amber, however they are typically implied. * **Variable Initialization**: Although the majority of contemporary C compilers automatically initialize variables to zero, the C specification states that variables should instead get the value currently stored in memory. This behavior may result in variables having unpredictable values when not explicitly initialized. Values in Amber are consistently initialized to zero by default. This approach enhances memory safety by ensuring that variables, including pointers, start with a known and safe initial value. This predictability is particularly important when working with pointers, as it reduces the risk of undefined behavior and contributes to safer memory management. * **Memory Management:** In C, memory management is manual, and developers must explicitly allocate and deallocate memory using functions like malloc and free. This can lead to memory leaks and pointer-related bugs if not handled carefully. Contrary to C, Amber includes automatic memory management through garbage collection, reducing the risk of memory-related errors. * **Error Handling and Exit Codes**: C traditionally uses error codes or return values for error handling, which can lead to verbose error-checking code. Whereas, Amber has a systematic approach to error handling using the **error** type and built-in mechanisms like **panic** and **recover**. Amber's main function consistently returns with the exit code 0. Call os if you want to return a different value. Where n is generally 1 to signal an error, use exit(n). To end the program, call this from any location—not just main. The exit(n) function, which is defined in stdlib.h, can be used in C to accomplish a similar functionality. |  |

**Advantages / Disadvantages (in comparison with C)**

|  |  |  |
| --- | --- | --- |
|  | **Advantage**:   * **Easy to learn with simpler Syntax –** Allows for more concise code that is easier to read and learn, for example, the elimination of semicolon at the end of every statement * **Memory management** - automatic memory management through garbage collection, reducing the risk of memory-related errors and simplifying memory allocation and deallocation.   **Error Handling** - systematic approach to error handling using the error type and built-in mechanisms like panic and recover. In comparison to C, which utilizes error codes or return values, making error-checking more tedious and difficult.  **Disadvantage**:   * **Runtime overhead –** including garbage collection, which can lead to some runtime overhead compared to C programs. This overhead may not be suitable for extremely low-latency or real-time systems where predictable execution times are crucial * **Less Control Over Memory** - less fine-grained control over memory allocation and deallocation compared to C. In performance-critical scenarios or resource-constrained environments, this may be a drawback. * **Smaller Language Feature Set** – Amber is intentionally designed to maintain simplicity and readability. However, this may limit its expressiveness in certain advanced programming scenarios where C's more extensive feature set might be necessary. * **No Direct Memory Pointer -** Amber intentionally restricts direct memory pointer arithmetic for safety reasons. While this enhances safety, it can be a limitation when working on certain low-level tasks where pointer arithmetic is required. * **Limited support for generics** - This limitation could lead to code duplication and less flexibility when writing code that needs to work with different data types. |  |

**Comparing with another language**

**Language Name: Java**

**Differences**

|  |  |  |
| --- | --- | --- |
|  | **Concurrency**:  Java: Java relies on threads and locks for concurrency. It has a more traditional threading model.  Amber: Amber uses Amber-routines and channels for concurrency. Amber-routines are lightweight threads, and channels facilitate communication between Amber-routines. Amber's concurrency model is designed to be simpler and more efficient than traditional threading.  **Typing**:  Java: Java is both statically and strongly typed. It requires explicit type declarations and type checking at compile time.  Amber: Amber is also statically typed and strongly typed, but it allows for type inference in some cases, reducing the need for explicit type declarations.  **Syntax:**  Java: Java has a more verbose syntax compared to Amber, with more boilerplate code required for common tasks.  Amber: Amber has a concise and minimalistic syntax, which promotes readability and simplicity.  **Object-Oriented vs. Simplicity:**  Java: Java is a class-based, object-oriented language that encourages the use of inheritance, encapsulation, and polymorphism.  Amber: Amber promotes a more composition-based approach to code reuse instead of relying heavily on inheritance. It prioritizes simplicity and ease of use.  **Error Handling:**  Java: Java uses exceptions for error handling, which can sometimes lead to complex code.  Amber: Amber uses explicit error values (usually returned as a tuple with the result), which can lead to more straightforward error handling.  **Garbage Collection:**  Java: Java has a garbage collector that manages memory automatically.  Amber: Amber also has a garbage collector, but it is designed to minimize latency spikes, making it more suitable for real-time applications.  **Ecosystem and Libraries:**  Java: Java has a vast and mature ecosystem with a wide range of libraries, frameworks (e.g., Spring, Hibernate), and tools.  Amber: Amber's ecosystem is growing but is still not as extensive as Java's. It is known for its simplicity and a "batteries-included" standard library.  **Cross-Platform Support:**  Java: Java applications are known for their "write once, run anywhere" capability, thanks to the Java Virtual Machine (JVM).  Amber: Amber also supports cross-platform development and can compile code for various operating systems and architectures.  **Generics:**  Java: Java has had generics since Java 5, allowing developers to write more type-safe and reusable code.  Amber: Amber has limited support for generics, leading to less flexibility when writing code that needs to work with different data types.  In summary, Amber and Java have different design philosophies and strengths, which make them suitable for different use cases. Amber is developed for simplicity and efficiency in concurrent programming, while Java's maturity and extensive ecosystem make it a strong choice for various application domains, particularly in enterprise software development. The choice between them should be based on project requirements and developer preferences. |  |

**Advantages / Disadvantages (in comparison with this second language)**

|  |  |  |
| --- | --- | --- |
|  | **Advantage**   1. **Concurrency Support**: Amber was designed with built-in support for concurrency through Amber-routines and channels. This makes it easier to write concurrent and parallel code compared to Java, which relies on threads and locks. 2. **Strongly Typed and Statically Typed**: Amber is statically typed, which means that many errors can be caught at compile time, reducing the likelihood of runtime errors compared to Java, which is also statically typed but has a more complex type of system. 3. **Simplicity and Readability**: Amber is known for its simplicity and readability. It has a clean and minimalistic syntax, which can be easier for developers to understand and maintain compared to Java's more verbose syntax. 4. **Fast Compilation**: Amber's compilation process is typically faster than Java's, which can lead to quicker development cycles. 5. **Garbage Collection**: Amber has a garbage collector that helps manage memory efficiently, making it easier for developers to write memory-safe code. While Java also has garbage collection, Amber's garbage collector is designed to minimize latency spikes, making it more suitable for real-time applications. 6. **Small Binary Size**: Amber compiles to a single binary file with minimal dependencies, making it suitable for creating lightweight and portable applications. 7. **Cross-Platform**: Amber supports cross-platform development and can compile code for various operating systems and architectures without much effort.   **Disadvantage**   1. **Less Mature Ecosystem**: Java has been around for a long time, and as a result, it has a vast and mature ecosystem with a wide range of libraries, frameworks, and tools. Amber's ecosystem is growing but is still not as extensive as Java's. 2. **No Inheritance**: Amber uses composition instead of inheritance for code reuse, which can be a departure from the traditional object-oriented approach used in Java. Some developers may find this design paradigm less intuitive. 3. **Less Enterprise Adoption**: Java is a dominant language in many large enterprises, making it easier to find Java developers and support for Java-based systems. 4. **Less Tooling**: Java has a vast array of IDEs, build tools, and debugging tools, while Amber's tooling is more minimalistic in comparison. |  |

|  |  |
| --- | --- |
| **Part**  **3** | **Architectural Questions** |

**Advantages**

The purpose of Amber is developed with the goal of being a simple language that can be a powerful tool for new programmers with its readability and efficiency. The language is designed to combine the minimalistic syntax with powerful performance and concurrency-support required to build modern, robust software systems.

**Strategy: C Implementation**

* The language will follow the sizes for data types and any restrictions, as seen implemented in C.
* Since the compiler will be implemented using ANSI C language, definitions such as: char datatypes using one single byte (i.e. Unicode cannot be accepted) will be respected, and if needed proper (new) structs can be used to simulate datatypes that do not exist originally.

***Note 1: C Datatypes***

*Remember that you are implementing your language in ANSI C. For this reason, you cannot create arbitrarily your language (from scratch). You need to use what is already provided by C Compiler. For this reason, think about using and defining the language obeying the datatypes.*

The compiler will follow a multi-step process of lexical analysis/tokenization, which involves breaking down the source code into smaller units called tokens.

1. **Lexical Analysis**: Breaks down the source code into tokens, identifying keywords, identifiers, literals, operators, and punctuation. This phase involves scanning the code character by character, and identify source code components into tokens.
2. **Token Classification**: Categorizes tokens into types such as keywords, identifiers, literals, operators, and punctuation.
3. **Symbol Table**: Maintains a data structure to track identifiers and their attributes, facilitating scope resolution and type checking.
4. **Syntax Analysis**: Analyzes the arrangement of tokens according to the language's grammar rules, creating a parse tree or abstract syntax tree (AST).
5. **Semantic Analysis**: Ensures the code's meaning is correct by performing checks like type checking, scope analysis, and variable declaration verification.
6. **Intermediate Code Generation**: Produces an intermediate representation of the code for optimization and translation to lower-level code.
7. **Optimization**: Applies transformations to improve code efficiency, including reducing redundant operations and optimizing memory usage.
8. **Code Generation**: Translates the optimized code or intermediate representation into executable code, such as machine code or bytecode, for execution.

* The tokenization will allow the compiler to understand and correctly translate the code into the desired output.
* The Output to console command will be enclosed in the Print() statement, with the contents to be output enclosed inside the brackets (). Literal texts are enclosed in double quotes (“”) and values stored in variables can be concatenated using the + operator.
* The Scope will be determined using the curly braces { } as delimiters, to explicitly define the beginning and end of a block of code.

**Git**

Link to GitHub: <https://github.com/philvilla25/compilerProject.git>

**References**

1. <https://go.dev/tour/basics/11>
2. <https://www.w3schools.com/go/go_syntax.php>

Algonquin College

Fall, 2023