Compilers - Assignment 01

HolyScript

Yash Kokane	**20110237**
Shah Faisal Khan	**21110156**
Aishwarya Omar	**20110008**
Rishabh Patidar	**20110165**

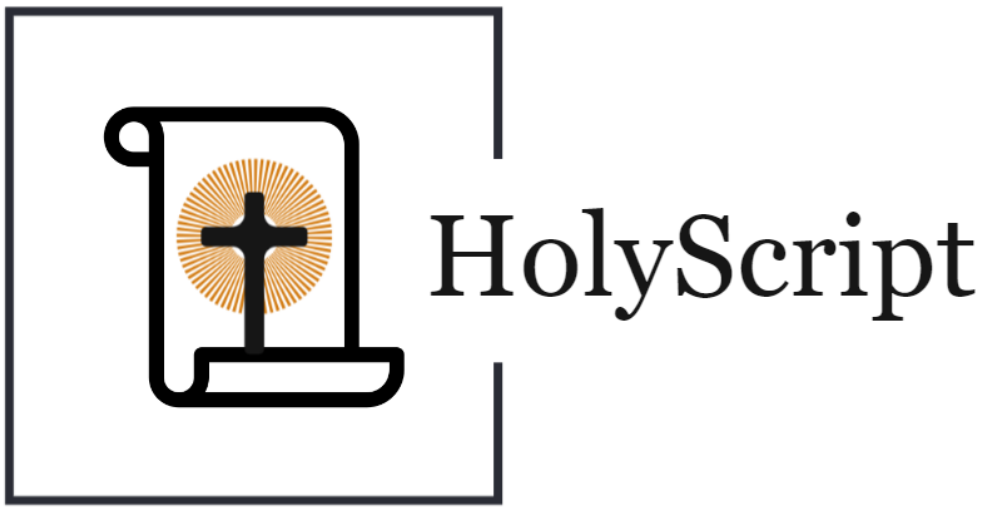

Table of Contents

Basic Syntax:

HolyScript is a strongly typed programming language with a unique thematic approach, incorporating biblical terms in its syntax. The language includes belief (if) and else statements, chant (for) loops, pledge (while) loops, and oath (do-while) loops, offering powerful tools for controlling program flow. HolyScript includes a try-catch mechanism, using the keywords trial and mercy. With the invoke keyword, HolyScript allows for defining functions. The language supports various data structures, including lists, arrays, and tuples.

Program Structure

- Start of Program: Every HolyScript program starts with the keyword summon followed by HolyScript. This signals the beginning of the code that will be considered for compilation.
- End of Program: Every HolyScript program ends with the keyword doom. This signifies the end of the code to be compiled.
- Comments: Any text outside the summon HolyScript and doom keywords, or any text followed by a double slash (//) is considered a comment and will not be executed or compiled. HolyScript supports only single line commenting.
- Curly Braces: Curly braces ({}) are used to define the scope of code blocks. They are commonly used in structures like function definitions, loops, and conditionals to group multiple statements together.
- Semicolons: In HolyScript, semicolons (;) are used to mark the end of a statement. It is a delimiter that separates different commands or expressions from each other.
- Indentation: Unlike Python, HolyScript does not use indentation to determine the scope or grouping of code. However, consistent indentation is encouraged for better readability and maintainability of code.
- File Extension: HolyScript files are saved with the .holy extension.

Example: General control structure for an example file, file.holy

```
summon HolyScript
    int x = 10; // variable declaration
    belief (x > 0) {
        preach("x is positive");
    } else {
        preach("x is not positive");
    }
doom
This will be a comment and will be ignored, since it is outside
the summon...doom block.
```

Keywords in HolyScript

Keyword	Description
`summon`	Begins a HolyScript program.
`doom`	Ends a HolyScript program.
`eternal`	Declares a constant.
`belief`	Begins an if-statement.
`else`	Specifies the alternative block of code for an if-statement.
`chant`	Begins a for-loop.
`pledge`	Begins a while-loop.
`oath`	Begins a do-while loop.
`preach`	Prints a value to the screen.
`invoke`	Defines a function.
`deliver`	Returns a value from a function.
`persist`	Continues to the next iteration of a loop.
`retreat`	Exits a loop prematurely.
`trial`	Begins a try block for error handling.
`mercy`	Begins a catch block for error handling.

`condemn`	Signals an error occurrence.
`unite`	Concatenates two strings.
`length`	Returns the number of elements or characters.
`head`	Retrieves the first element of a lis.
`tail`	Returns a new list excluding the first element.
`cons`	Adds a new element to the beginning of a list.
`append`	Adds an element to the end of a list or array.
`insert`	Inserts an element at a specified index in a list.
`remove`	Deletes an element by value from a list.
`slice`	Extracts a substring or subarray.

Basic Types and Operations:

Representation

Integers (int)

Whole numbers without a fractional component.

Size and Range: 4 bytes; from -2,147,483,648 to 2,147,483,647.

Example: int x = 42;

Floating-Point Numbers (float)

Numbers that include a decimal point for representing fractions and more precise measurements.

Size and Range: 4 bytes; approximately 1.4E-45 to 3.4E+38.

Example: float pi = 3.14;

Double Precision Floating-Point Numbers (double)

Similar to float but with double the precision.

Size and Range: 8 bytes; approximately 4.9E-324 to 1.8E+308.

Example: double precisePi = 3.1415926535;

Characters (char)

Represents a single character.

Size and Range: 1 byte; from 0 to 255 (unsigned) or -128 to 127 (signed).

Example: char initial = 'H';

Booleans (bool)

Represent truth values, used for logical operations and control flow.

HolyScript defines two boolean values:

truth: Represents a true value.

myth: Represents a false value.
Example: bool isReady = truth;

Arithmetic Operations

HolyScript supports a variety of arithmetic operations on numbers, which are essential for calculations and numerical manipulations.

- Addition (+): int sum = 5 + 3;
- Subtraction (-): int difference = 10 - 4;
- Multiplication (*): int product = 7 * 3;
- Division (/): float quotient = 10 / 4.0;
- Modulo (%): int remainder = 10 % 3; (Valid only for integers)

Unary Operations

Unary operations in HolyScript involve a single operand and are used to modify the value of a number.

- Negation (-): int negated = -5;
- Increment (++): int count = 0; count++;
- Decrement (--): int count = 10; count--;

Special Numbers

HolyScript also includes special numbers that are used in specific contexts or to represent certain unique values.
infinity: Represents an infinitely large number. It is typically the result of dividing a number by zero or an operation that exceeds the maximum representable value. Example: int inf = 1 / 0; // inf is infinity
NaN (Not a Number): Stands for "Not a Number" and is used to represent a value that is not a valid number. This is usually the result of undefined or erroneous mathematical operations. Example: bless nan = 0 / 0; // nan is NaN

Logical Operations

HolyScript supports logical operations on boolean values, allowing complex conditions and decision-making processes.

AND (&&): Returns truth if both operands are truth; otherwise, returns myth.

- Example: bool result = truth && myth; // result is myth

OR (||): Returns truth if at least one operand is truth; otherwise, returns myth.

- Example: bool result = truth || myth; // result is truth

NOT (!): Inverts the boolean value; truth becomes myth and vice versa.

- Example: bool result = !truth; // result is myth

Comparisons

Booleans are often the result of comparison operations. HolyScript provides several relational operators to compare values:

Less Than (<): Returns truth if the first value is less than the second.

- Example: bool isLess = 3 < 5; // isLess is truth

Greater Than (>): Returns truth if the first value is greater than the second.

- Example: bool isGreater = 10 > 7; // isGreater is truth

Equal To (==): Returns truth if both values are equal.

- Example: bool isEqual = 4 == 4; // isEqual is truth

Not Equal To (!=): Returns truth if the values are not equal.

- Example: bool isNotEqual = 4 != 5; // isNotEqual is truth

Less Than or Equal To (<=): Returns truth if the first value is less than or equal to the second.

- Example: bool isLessOrEqual = 3 <= 3; // isLessOrEqual is truth

Greater Than or Equal To (>=): Returns truth if the first value is greater than or equal to the second.

- Example: bool isGreaterOrEqual = 5 >= 4; // isGreaterOrEqual is truth

Boolean Context

In certain situations, values of other types are evaluated in a boolean context. For example, in conditional statements, non-boolean values need to be interpreted as either truth or myth.

Numbers: By convention, 0 is considered myth, and all other numbers are considered truth.

Strings: An empty string ("") is considered myth, while any non-empty string is considered truth.

Strings:

Representation

In HolyScript, strings are sequences of characters used to represent text. They are enclosed in either single (' ') or double (" ") quotes. Strings can contain letters, digits, symbols, and whitespace characters.

Examples:

- Single-quoted: 'Hello, HolyScript!'
- Double-quoted: "Learning HolyScript is fun!"

Concatenation

The unite function in HolyScript concatenates two strings, creating a new string that is the combination of both.

Syntax: unite(string1, string2)

Example:

```
str greeting = "Hello, ";
str name = "Alice";
str fullGreeting = unite(greeting, name);  // "Hello, Alice"
```

Slicing

The slice function allows you to extract a substring from a string, based on the start and end indices.

Syntax: slice(string, start, end)

Example:

```
str text = "HolyScript";
str subtext = slice(text, 1, 4);  // "olyS"
```

Character Access

In HolyScript, you can access individual characters in a string using the square bracket notation, similar to accessing elements in an array.

Syntax: string[index]

Example:

```
str text = "HolyScript";
str firstChar = text[0];  // "H"
str thirdChar = text[2];  // "l"
```

String Length

The length function in HolyScript returns the number of characters in a string.

Syntax: length(string)

Example:

```
str text = "HolyScript";
int size = length(text);  // 10
```

Compound Types

Definition

Tuples in HolyScript are immutable, dynamic collections that can store a fixed number of elements of different types in a specific order.

Syntax: tuple tupleName = (value1, value2, ...);

Example: tuple myTuple = (1, "hello", 3.14, truth);

Properties

Immutable: Once created, elements cannot be modified.

Ordered: Elements are accessed by their index.

Heterogeneous: Supports elements of different types.

Basic Functions

Accessing Elements

Elements in a tuple can be accessed using indexing. Indexing starts at 0.

Syntax: type element = myTuple[index];

Example:int firstElement = myTuple[0]; // Accesses the first element (1)

Length

The length function returns the number of elements in the tuple.

Syntax: length(myTuple)

Example: int size = length(myTuple); // 4

Iteration

Elements of a tuple can be iterated over using the chant loop.

```
chant (int i = 0; i < length(myTuple); i++) {
    preach(myTuple[i]);
}
```

Use Cases

Tuples are particularly useful for:

- Storing a collection of related but different type elements, such as a coordinate point (x, y, z).
- Returning multiple values from a function.
- Ensuring that certain data remains unchanged throughout the program.

Lists:

Lists in HolyScript are dynamic collections that can contain elements of different types. This allows for a high degree of flexibility in storing and manipulating collections of data, where the type of each element is inferred by the language.

Definition

Lists are defined using square brackets [] and can contain any number of elements, possibly of different types.

Syntax: [element1, element2, element3, ...]
Example: list myList = [1, "hello", 3.14, truth];

Properties

- Type Inferencing: The type of each element in the list is inferred by HolyScript, allowing for heterogeneous collections.

- Mutable: Elements in a list can be added, removed, or changed, offering flexibility in managing the collection.
- Ordered: The elements in a list are indexed and can be accessed or modified in the order they were added.

Basic Operations

Length: Returns the number of elements in the list.

Syntax: length(myList)

Example: int size = length(myList); // size is the number of elements in myList

Head: Retrieves the first element of the list.

Syntax: head(myList)

Example: var firstElement = head(myList); // firstElement is the first item in myList

Tail: Returns a new list containing all elements except the first.

Syntax: tail(myList)

Example: var restElements = tail(myList); // restElements contains all but the first item of myList

Cons: Adds a new element to the beginning of the list.

Syntax: cons(element, myList)

Example: var newList = cons(newElement, myList); // newList starts with newElement followed by all elements of myList

Mutable Operations

Adding Elements

Append: Adds an element to the end of the list.

Syntax: myList.append(element);

Example: myList.append("new"); // Adds "new" to the end of myLis

Insert: Inserts an element at a specified index.
Syntax: myList.insert(index, element);
Example: myList.insert(2, "inserted"); // Inserts "inserted" at index 2

Removing Elements

Use remove to delete an element by index.
Syntax: remove(myList, index)
Example: remove(myList, 1); // myList becomes [1, 3.14, truth]

Iteration

Use the chant loop for iterating over elements in the list.
Example:
holy
Copy code

```
chant (int i = 0; i < length(myList); i++) {
    preach(myList[i]);
}
```

Use Cases

Lists are suitable for:

- Storing and manipulating a collection of items where the number of elements can change.
- Implementing data structures like stacks, queues, or linked lists.
- Performing operations that require frequent addition or removal of elements.

Arrays:

Overview

Arrays in HolyScript are dynamic, strongly typed collections designed to store multiple items of the same type. Unlike fixed-size arrays in some languages, HolyScript arrays can grow or shrink in size, offering flexibility similar to lists but with the performance benefits and type safety of arrays.

Definition

Arrays must be declared with a specific element type, and they can be resized dynamically to accommodate more elements or reduce their size.

Syntax: array<elementType> arrayName = [element1, element2, ...];
Example: array<int> myarray = [10, 20, 30];

Basic Operations

Arrays share many operations with lists, such as **length and head.** Additional array-specific operations also exist.

Length: Returns the number of elements in the array.
Syntax: length(myArray)
Example: int size = length(myArray); // size is the number of elements in myArray

Head: Retrieves the first element of the array.
Syntax: head(myArray)
Example: var firstElement = head(myArray); // firstElement is the first item in myArray

Append: Automatically adjusts the array's size.

Syntax: arrayName.append(element);
Example: myarray.append(40); // myarray becomes [10, 20, 30, 40]

Remove: Adjusts the size accordingly.
Syntax: arrayName.remove(index);
Example: myarray.remove(0); // myarray becomes [20, 30, 40]

Accessing Elements: By index.
Syntax: elementType element = arrayName[index];
Example: int firstElement = myarray[0]; // 20

Iteration
Similar to lists, use the chant loop for iterating over elements in the array.

```
chant (int i = 0; i < length(myArray); i++) {
   preach(myArray[i]);
}
```

Use Cases

Arrays are particularly useful for:

- Storing and processing large amounts of numerical data.
- Implementing data structures where the size is known in advance or does not change frequently.
- Situations requiring performance optimization, especially in mathematical and scientific computations.

Conditionals

in HolyScript are used to execute different blocks of code based on specific conditions. They are crucial for controlling the flow of the program and making

decisions. HolyScript uses the keywords belief for if-statements and else for else-statements.

Boolean Expressions: Conditions in belief statements should be boolean expressions.

Braces: Use curly braces {} to define the scope of each conditional block.

Readability: While HolyScript does not use indentation for scoping, consistent indentation is recommended for clarity.

If-Else Statement

The belief statement executes a block of code if a specified condition is true (truth). If the condition is false (myth), the code inside the else block is executed.

Syntax:

```
belief (condition) {
   // code to execute if condition is truth
} else {
   // code to execute if condition is myth
}
```

Example:

```
int x = 10;
belief (x > 0) {
   preach("x is positive");
} else {
   preach("x is negative or zero");
}
```

Else-If Statement

The else belief construct allows for multiple conditions to be tested in sequence.

Syntax:

```
belief (condition1) {
   // code for condition1
```

```
} else belief (condition2) {
    // code for condition2
} else {
    // code if neither condition1 nor condition2
}
```

Example:

```
int x = 10;
belief (x > 0) {
    preach("x is positive");
} else {
    preach("x is negative or zero");
}
```

Nested Conditionals

Nested conditionals involve placing a conditional statement inside another conditional statement. This allows for more complex decision-making scenarios.

Example:

```
int a = 10, b = -5;
belief (a > 0) {
    belief (b > 0) {
        preach("Both a and b are positive");
    } else {
        preach("a is positive, b is not");
    }
} else {
    preach("a is not positive");
}
```

Loops

Loops in HolyScript are constructs used to execute a block of code repeatedly under certain conditions. HolyScript supports three main types of loops: chant (for loop), pledge (while loop), and oath-pledge (do-while loop).

For Loop (chant)

The chant loop is used for iterating a specific number of times or over a range of values.

Syntax:

```
chant (initialization; condition; update) {
    // code to be executed
}
```

Example:

```
chant (int i = 0; i < 5; i++) {
    preach(i);
}
```

This loop prints the numbers 0 to 4.

While Loop (pledge)

The pledge loop executes as long as a specified condition remains true.

Syntax:

```
pledge (condition) {
    // code to be executed while condition is truth
}
```

Example:

```
int x = 5;
pledge (x > 0) {
    preach(x);
    x--;
```

```
}
```

This loop prints the numbers 5 to 1.

Do-While Loop (oath-pledge)

The oath loop ensures that the code block is executed at least once before checking the condition.

Syntax:

```
oath {
   // code to be executed
} pledge (condition);
```

Example:

```
int y = 5;
oath {
   preach(y);
   y--;
} pledge (y > 0);
```

Loop Control Statements

persist (Continue): The persist statement is used to skip the remaining code in the current iteration of the loop and proceed to the next iteration. It's particularly useful when you want to avoid executing some part of the loop under certain conditions.

Example with chant Loop:

```
chant (int i = 0; i < 10; i++) {
   belief (i == 5) {
      persist;
   }
```

```
    preach(i);
}
```

In this example, when i equals 5, the persist statement is executed, which skips the preach(i) statement for that iteration only. The loop will print all numbers from 0 to 9 except for 5.

retreat (Break): The retreat statement is used to exit a loop immediately, regardless of the loop's condition. It's useful for stopping the loop execution when a certain condition is met.

Example with pledge Loop:

```
int x = 0;
pledge (truth) {
    x++;
    belief (x == 5) {
        retreat;
    }
    preach(x);
}
```

In this example, the loop is an infinite loop (pledge (truth)), but it will exit when x reaches 5 due to the retreat statement. The loop will print numbers 1 to 4, and then exit when x is 5.

Functions

Defining Functions

In HolyScript, functions are defined with a clear specification of their return type, followed by the keyword invoke, the function name, and a list of typed parameters.

The body of the function is enclosed within curly braces {}. To return a value, the deliver keyword is used.

Syntax:

```
invoke returnType functionName(type parameter1, type parameter2, ...) {
    // function body
    // use 'deliver' to return a value
}
```

Example:

```
invoke int addNumbers(int a, int b) {
    invoke int sum = a + b;
    deliver sum;
}
```

Calling Functions

Functions are called by specifying the function name followed by a list of arguments enclosed in parentheses.

Example:

```
summon HolyScript
    int result = addNumbers(5, 7);
    preach(result); // Outputs: 12
doom
```

Return Values

Functions can return a value using the deliver keyword. If no deliver is used, the function does not return a value.

Example:

```
str invoke greet(name) {
    str greeting = unite("Hello, ", name);
    deliver greeting;
}
str message = greet("Alice"); // message is "Hello, Alice"
```

Inbuilt Functions

The preach (print) Function

The preach function in HolyScript is a built-in function used for outputting values to the screen. It plays a crucial role in displaying information, debugging, and providing feedback to the user.

Syntax:

The preach function takes a single argument, which is the value to be printed. This value can be of any type, such as a number, string, boolean, or even complex data structures like lists or tuples.

Syntax: preach(value);

Examples

Printing a String:

```
str message = "Hello, HolyScript!";
preach(message);  // Outputs: Hello, HolyScript!
```

Printing a Number:

```
str number = 42;
preach(number);  // Outputs: 42
```

Printing a Boolean:

```
bool condition = truth;
```

```
preach(condition);  // Outputs: truth
```

Printing a List:

```
bless myList = [1, 2, 3, "holy"];
preach(myList);  // Outputs: [1, 2, 3, "holy"]
```

Closure

- A closure is created when a function is defined within another function, and the inner function accesses variables from the outer function. The inner function retains access to these variables even after the outer function has finished executing.
- The scope of an object written inside curly brackets is within the brackets only.
- The scope of an object written outside is everywhere in the code, i.e., global scope.

Example of Closures in HolyScript:

```
invoke str outerFunction() {
   str outsideVariable = "Holy";

   invoke str innerFunction() {
      str insideVariable = "Script";
          str combined = unite(outsideVariable, insideVariable); // Combine and assign to variable
      preach(combined); // Optionally preach the combined string
      deliver combined; // Use the combined string as the return value
   }

   deliver innerFunction; // Deliver the inner function as a closure
```

```
}

invoke str myClosure = outerFunction(); // myClosure now holds the closure
myClosure(); // Executes the closure, Outputs: HolyScript
```

Exception handling

Try-catch:

Similar to C++, we have try-catch block for exception handling where it will print something instead of crashing the program:

Syntax Overview:

- trial: Used to begin a block where exceptions might occur.
- condemn: Signals that an error has occurred, effectively raising an exception.
- mercy: Catches the error raised by condemn and allows for error handling.

Example:

```
summon HolyScript
int divisor = 0;
int result;
trial {
    if (divisor == 0) {
        condemn "Division by zero attempted.";
    } else {
        result = 10 / divisor;
        preach("Result: " + result);
    }
} mercy {
    preach("Runtime Error: Division by zero");
```

```
}
doom
```

Error Handling in HolyScript:

When there are bugs in the code, the program may encounter errors that cause the program to stop running. These errors can be accompanied by an error message that provides information about the cause and location of the error.

Type Error

Type errors arise when an operation or function is applied to a variable of an incompatible type, leading to an inconsistency in data handling.

Example:

```
summon HolyScript
   str x = "ten";
   int y = x + 2;
doom
```

Output:

```
Type Error: A square peg for a round hole, it seems.
Attempted to perform arithmetic on a string 'x' at line 3.
```

Syntax Error

Syntax errors occur when the program violates the grammatical rules of HolyScript, preventing the interpreter from parsing and executing the code.

Example:

```
summon HolyScript
   int x = 10
   belief(x > 0) {
      preach("x is positive")
   // Missing closing brace
doom
```

In this example, a missing closing brace leads to a syntax error.

Output:

Syntax Error: As a feast without food, so is thy code without sense.

Unexpected end of input, expecting '}' at line 4.

Runtime Error

Runtime errors occur during the execution of a program due to invalid operations, out-of-bound references, or other unforeseen issues that disrupt the normal flow of execution.

Example:

```
summon HolyScript
   int x = 10;
   int y = 0;
   int z = x / y;
doom
```

Output:

Runtime Error: Thy path was clear, yet how hast thou stumbled?

Division by zero attempted at line 4.