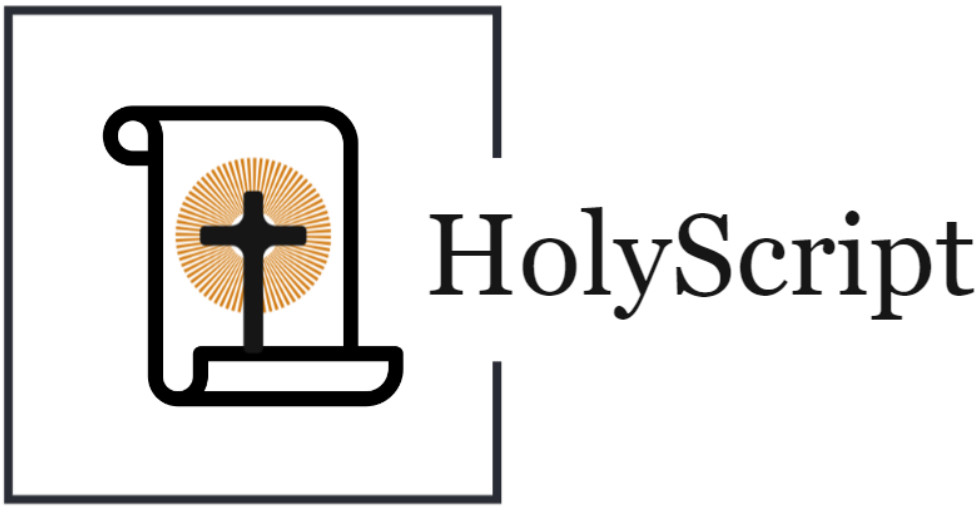
Compilers - Assignment 02

HolyScript



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

Token Classes in HolyScript:

Whitespace
Regular Expression: [\t\r\n\f\v]+
Explanation: Whitespace characters, such as spaces, tabs, and line breaks, are used to separate tokens and format the code. They are generally **ignored by the interpreter** except as separators.

Integer
Regular Expression: -?[0-9]+
Explanation: An integer in HolyScript can be positive or negative and is composed of a sequence of digits (0-9). If negative, it is prefixed with a minus sign (-).

Float
Regular Expression: -?[0-9]+\.[0-9]+
Explanation: A float represents a decimal number, consisting of one or more digits both before and after the decimal point (.). Negative floats are prefixed with a minus sign.

String
Regular Expression: "(?:[^"\\]|\\.)*"
Explanation: Strings are enclosed in double quotes and can include any character. Escape sequences are introduced with a backslash (\), allowing special characters within the string.

Keyword
Regular Expression:
be|eternal|belief|else|chant|pledge|oath|preach|invoke|deliver|persist|retreat|trial|mercy|condemn|unite|slice
Explanation: Keywords are reserved words that have special meanings in HolyScript, defining control structures, operations, and language constructs.

Boolean
Regular Expression: myth|truth
Explanation: Booleans represent logical values in HolyScript, with truth indicating true and myth indicating false.

Type Keyword
Regular Expression: int|float|char|bool|str
Explanation: Type keywords specify the data type of variables in HolyScript, including integers, floating-point numbers, characters, booleans, and strings
.
Operator
Regular Expression: [\+\-*/%=\(\)\[\]\{\},;:?!]

Explanation: Operators include symbols for mathematical, logical, and assignment operations, as well as punctuation for code structure and statement separation.

Comment
Regular Expression: //.*
Explanation: Comments begin with // and extend to the end of the line, allowing for annotations or explanations in the code that are ignored during execution. These are **ignored by the lexer**.

End of Statement
Regular Expression: ;
Explanation: The semicolon marks the end of a statement, allowing for the separation of multiple statements on a single line.

List Utilities
Regular Expression: (cons|head|tail|append|insert|remove|length)
Explanation: This pattern matches any of the specified list utility function names. These names are used for operations that manipulate list data structures, allowing for adding, removing, accessing, or determining the size of lists.

Array Utilities
Regular Expression: (head|append|remove|length)
Explanation: Matches function names dedicated to array manipulation. Given arrays might have slightly different allowed operations than lists (considering language design decisions on mutability and type constraints), this regex covers common array operations.

Tuple Utilities
Regular Expression: (length)
Explanation: Since tuples are immutable and their primary queryable attribute is their size, the length function is the only utility included here. This pattern matches the length keyword used to determine the number of elements in a tuple.

Note: The statements **summon HolyScript** and **doom** serve a unique function distinct from the conventional tokens typically processed by the lexer. While these statements act as program delimiters, marking the beginning and end of a HolyScript script respectively, they are not directly tokenized or processed by the lexer as part of the lexical analysis. Instead, their primary purpose is to delineate the scope of the script—indicating precisely which portion of the text should undergo lexical analysis.

Classes defined in lexer.py:

Errors

Error: The base error class encapsulates general error information, including the position (line and column) in the source code where the error occurred. This class can be extended to create more specific error types.

IllegalCharError: Inherits from the Error class, specifically designed to report instances where an unexpected or unsupported character is encountered in the source code.

Position

This class is crucial for error reporting and debugging, as it helps pinpoint the exact location of tokens and errors within the source code. It keeps track of the current index, line number, column number, and also stores the filename and the full text of the file being parsed. This enables precise error messages and facilitates understanding of the source code's structure.

Tokens

The lexer uses various data classes to represent tokens, each tailored to a specific type of lexical unit in HolyScript:

- **Int, Float, Bool, CharToken:** Represent integer, floating-point numbers, boolean values, and characters, respectively.
- **EndOfStatement**: Represents the end of a statement, typically denoted by a semicolon (;).
- **Keyword, TypeKeyword:** Represent reserved words in HolyScript that have special meaning, with TypeKeyword specifically for type annotations (e.g., int, float).
- **Identifier**: Represents names given to variables, functions, etc.
- **Operator, Symbols:** Represent operational and symbolic tokens used for expressions and structure within the code (e.g., +, -, (,)).
- **StringToken, ListToken:** Represent string literals and list structures.
- Whitespace: Represents spaces, tabs, and newlines, generally ignored but crucial for separating tokens.
- **UtilityFunction:** Represents functions for operating on data structures like lists, arrays, and tuples (e.g., head, tail).

Lexer

The Lexer class is tasked with converting a sequence of characters (the source code) into a sequence of tokens. This process, known as lexical analysis or tokenization, is foundational for the parsing phase.

Key Methods:

- **make_tokens:** The core method of the Lexer class, responsible for iterating over the source code and generating a list of tokens. It utilizes the advance, peek, and specific token creation methods (make_string, make_number, etc.) to identify and classify each part of the source code. This method carefully handles different token types, including handling edge cases like distinguishing between operators and symbols or between keywords and identifiers. The result is a sequence of tokens that accurately represents the syntactical elements of the source code, ready for parsing.
- **advance:** Moves the current position forward by one character, updating line and column numbers as needed. It's essential for progressing through the source text.
- **peek:** Looks at the next character without advancing the current position. This is useful for lookahead operations where decision-making depends on subsequent characters.
- **generate_token_mappings:** Initializes mappings that relate specific character combinations to token types. This includes both single-character tokens (e.g., (,)) and potentially double-character tokens (e.g., &&, ||).
- **skip_comment:** When a comment start sequence is detected, this method advances the lexer's position past the comment, effectively ignoring the comment content.
- **make_string:** Constructs a StringToken when a string literal start (e.g., a double quote) is encountered, handling escape sequences and closing quotes.
- **make_char:** Similar to make_string, but for character literals, typically enclosed in single quotes.
- **extract_identifier:** Gathers a sequence of characters that form an identifier, stopping at whitespace or symbols that cannot be part of an identifier.
- **make_identifier:** Determines if the extracted sequence is a keyword or a user-defined identifier, creating the appropriate token type.
- **make_number:** Handles numeric literals, distinguishing between integers and floats based on the presence of a decimal point.

Instructions for obtaining lexer output:

Through script:

- Prepare Your HolyScript File: Write your HolyScript code in a file with the .holy extension. Ensure it starts with summon HolyScript and ends with doom.
- Execute main.py: Run main.py with the path to your .holy file as an argument. Use the following command in your terminal or command prompt:

```
python main.py path/to/your_script.holy
```

- View Output: If your script is correctly formatted and contains no errors, the lexer output will be printed to the terminal, listing the tokens identified within your script.

Command Line Interface:

- Start the CLI: Run main.py without any arguments:
  ```
  python main.py
  ```
- This will launch the HolyScript CLI, welcoming you to input code directly.
- Enter Code: Type HolyScript code directly into the CLI. Press Enter to tokenize the current line.
- View Output: After each line is input, the lexer output (tokens) for that line will be printed immediately.
- Exit CLI: Type exit and press Enter to leave the CLI.

Lexer Outputs:

Testcase 1: sample1.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.
```

Output:

```
PS C:\Users\yashk\OneDrive\Documents\Projects\Compiler> python
main.py sample1.holy
TypeKeyword(value='int'), Identifier(name='x'),
Operator(value='='), Int(value=10), EndOfStatement(value=';'),
Keyword(value='belief'), Symbols(value='('),
Identifier(name='x'), Operator(value='>'), Int(value=0),
Symbols(value=')'), Symbols(value='{'), Keyword(value='preach'),
```

```
Symbols(value='('), (StringToken(value='x is positive'), None),
Symbols(value=')'), EndOfStatement(value=';'),
Symbols(value='}'), Keyword(value='else'), Symbols(value='{'),
Keyword(value='preach'), Symbols(value='('),
(StringToken(value='x is not positive'), None),
Symbols(value=')'), EndOfStatement(value=';'), Symbols(value='}')
```

Testcase 2: sample2.holy

Code:

```
summon HolyScript

// Declare and initialize an array of numbers
array<int> numbers = [1, 3, 5, 7, 9];

// Use a chant loop to iterate over the array and preach each
number
chant (int i = 0; i < length(numbers); i++) {
    preach(numbers[i]);
}

// Initialize a variable for the pledge-oath loop
int j = 0;

// Use a pledge-oath loop to find and preach the first number
greater than 5
oath {
    // Check if the current number is greater than 5
    belief (numbers[j] > 5) {
        preach(numbers[j]);
        retreat; // Exit the loop
    }
    j++;
} pledge (j < length(numbers));

Doom
```

Output:

```
PS C:\Users\yashk\OneDrive\Documents\Projects\Compiler> python
main.py sample2.holy
Identifier(name='array'), Operator(value='<'),
TypeKeyword(value='int'), Operator(value='>'),
Identifier(name='numbers'), Operator(value='='),
Symbols(value='['), Int(value=1), Symbols(value=','),
Int(value=3), Symbols(value=','), Int(value=5),
Symbols(value=','), Int(value=7), Symbols(value=','),
Int(value=9), Symbols(value=']'), EndOfStatement(value=';'),
Keyword(value='chant'), Symbols(value='('),
TypeKeyword(value='int'), Identifier(name='i'),
Operator(value='='), Int(value=0), EndOfStatement(value=';'),
Identifier(name='i'), Operator(value='<'),
UtilityFunction(value='length'), Symbols(value='('),
Identifier(name='numbers'), Symbols(value=')'),
EndOfStatement(value=';'), Identifier(name='i'),
Operator(value='+'), Operator(value='+'), Symbols(value=')'),
Symbols(value='{'), Keyword(value='preach'), Symbols(value='('),
Identifier(name='numbers'), Symbols(value='['),
Identifier(name='i'), Symbols(value=']'), Symbols(value=')'),
EndOfStatement(value=';'), Symbols(value='}'),
TypeKeyword(value='int'), Identifier(name='j'),
Operator(value='='), Int(value=0), EndOfStatement(value=';'),
Keyword(value='oath'), Symbols(value='{'),
Keyword(value='belief'), Symbols(value='('),
Identifier(name='numbers'), Symbols(value='['),
Identifier(name='j'), Symbols(value=']'), Operator(value='>'),
Int(value=5), Symbols(value=')'), Symbols(value='{'),
Keyword(value='preach'), Symbols(value='('),
Identifier(name='numbers'), Symbols(value='['),
Identifier(name='j'), Symbols(value=']'), Symbols(value=')'),
EndOfStatement(value=';'), Keyword(value='retreat'),
EndOfStatement(value=';'), Symbols(value='}'),
Identifier(name='j'), Operator(value='+'), Operator(value='+'),
EndOfStatement(value=';'), Symbols(value='}'),
Keyword(value='pledge'), Symbols(value='('),
Identifier(name='j'), Operator(value='<'),
UtilityFunction(value='length'), Symbols(value='('),
```

```
Identifier(name='numbers'), Symbols(value=')'),
Symbols(value=')'), EndOfStatement(value=';')
```

Testcase 3: sample3.holy

Code:

```
summon HolyScript

invoke int calculateFactorial(int n) {
    // Check if n is 0 or 1, factorial is 1 in these cases
    belief (n == 0 || n == 1) {
        deliver 1;
    } else {
        int result = 1; // Initialize the result variable to 1
        chant (int i = 2; i <= n; i++) {
            result = result * i; // Multiply result by i
        }
        deliver result;
    }
}

int numberToCalculate = 5; // Change this to any positive integer

invoke int factorialResult =
calculateFactorial(numberToCalculate);

preach(factorialResult);

Doom
```

Output:

```
PS C:\Users\yashk\OneDrive\Documents\Projects\Compiler> python
main.py sample3.holy
Keyword(value='invoke'), TypeKeyword(value='int'),
Identifier(name='calculateFactorial'), Symbols(value='('),
TypeKeyword(value='int'), Identifier(name='n'),
```

```
Symbols(value=')'), Symbols(value='{'), Keyword(value='belief'),
Symbols(value='('), Identifier(name='n'), Operator(value='=='),
Int(value=0), Operator(value='||'), Identifier(name='n'),
Operator(value='=='), Int(value=1), Symbols(value=')'),
Symbols(value='{'), Keyword(value='deliver'), Int(value=1),
EndOfStatement(value=';'), Symbols(value='}'),
Keyword(value='else'), Symbols(value='{'),
TypeKeyword(value='int'), Identifier(name='result'),
Operator(value='='), Int(value=1), EndOfStatement(value=';'),
Keyword(value='chant'), Symbols(value='('),
TypeKeyword(value='int'), Identifier(name='i'),
Operator(value='='), Int(value=2), EndOfStatement(value=';'),
Identifier(name='i'), Operator(value='<='), Identifier(name='n'),
EndOfStatement(value=';'), Identifier(name='i'),
Operator(value='+'), Operator(value='+'), Symbols(value=')'),
Symbols(value='{'), Identifier(name='result'),
Operator(value='='), Identifier(name='result'),
Operator(value='*'), Identifier(name='i'),
EndOfStatement(value=';'), Symbols(value='}'),
Keyword(value='deliver'), Identifier(name='result'),
EndOfStatement(value=';'), Symbols(value='}'),
Symbols(value='}'), TypeKeyword(value='int'),
Identifier(name='numberToCalculate'), Operator(value='='),
Int(value=5), EndOfStatement(value=';'), Keyword(value='invoke'),
TypeKeyword(value='int'), Identifier(name='factorialResult'),
Operator(value='='), Identifier(name='calculateFactorial'),
Symbols(value='('), Identifier(name='numberToCalculate'),
Symbols(value=')'), EndOfStatement(value=';'),
Keyword(value='preach'), Symbols(value='('),
Identifier(name='factorialResult'), Symbols(value=')'),
EndOfStatement(value=';')
```

Testcase 4: sample4.holy

Code:

```
summon HolyScript
```

```
// Declare and initialize the tuple
tuple myTuple = (1, 2, 3, 4, 5);

// Iterate over the tuple elements
chant (let i = 0; i < length(myTuple); i++) {
    // Check if the current index is 3
    belief (i == 3) {
        retreat; // Break the loop if i is 3
    }
    // Preach (print) the current element of the tuple
    preach(myTuple[i]);
}

doom
```

Output:

```
PS C:\Users\yashk\OneDrive\Documents\Projects\Compiler> python
main.py sample4.holy
Identifier(name='tuple'), Identifier(name='myTuple'),
Operator(value='='), Symbols(value='('), Int(value=1),
Symbols(value=','), Int(value=2), Symbols(value=','),
Int(value=3), Symbols(value=','), Int(value=4),
Symbols(value=','), Int(value=5), Symbols(value=')'),
EndOfStatement(value=';'), Keyword(value='chant'),
Symbols(value='('), Identifier(name='let'), Identifier(name='i'),
Operator(value='='), Int(value=0), EndOfStatement(value=';'),
Identifier(name='i'), Operator(value='<'),
UtilityFunction(value='length'), Symbols(value='('),
Identifier(name='myTuple'), Symbols(value=')'),
EndOfStatement(value=';'), Identifier(name='i'),
Operator(value='+'), Operator(value='+'), Symbols(value=')'),
Symbols(value='{'), Keyword(value='belief'), Symbols(value='('),
Identifier(name='i'), Operator(value='=='), Int(value=3),
Symbols(value=')'), Symbols(value='{'), Keyword(value='retreat'),
EndOfStatement(value=';'), Symbols(value='}'),
Keyword(value='preach'), Symbols(value='('),
```

```
Identifier(name='myTuple'), Symbols(value='['),
Identifier(name='i'), Symbols(value=']'), Symbols(value=')'),
EndOfStatement(value=';'), Symbols(value='}')
```