## Task 1

***a. Advantages and Disadvantages of Using an Interpreter Versus a Compiler***

*Advantages*

**Interpreter**

An interpreter can generate an ***immediate level of feedback*** by executing code line by line. This can be determined as one of the most beneficial elements for students, as it helps them in the process of learning by testing their code in real-time execution. As opined by Wang (2024, p.15), it can be justified that an interpreter also provides an ***easier process of debugging*** by reporting all of the potential errors in an immediate manner. It can be stated that it also provides more ***flexibility*** by allowing a dynamic kind of typing as well as experimenting without any requirements for full compilation.

**Complier**

As opined by Wang (2021, p.8), it can be determined that a compiler provides a ***faster process of execution*** by producing optimized code for the machine. It can also be determined as a ***better approach for complex kinds of applications***. As it provides a more efficient procedure of scaling, while a large scale of application is compiled by the programmers. It also does ***not have any runtime dependencies***, which makes it more efficient and reliable for use in any aspect.

*Disadvantages*

**Interpreter**

As opined by Milewicz (2021, p.8), an interpreter can be determined as having a ***slower process of execution*** as it proceeds line by line, thereby increasing the overhead. It also provides

***limited capability for optimization***. Using an interpreter also imposes several issues regarding the process of ***distribution***, as it requires the installation of the interpreter in every system.

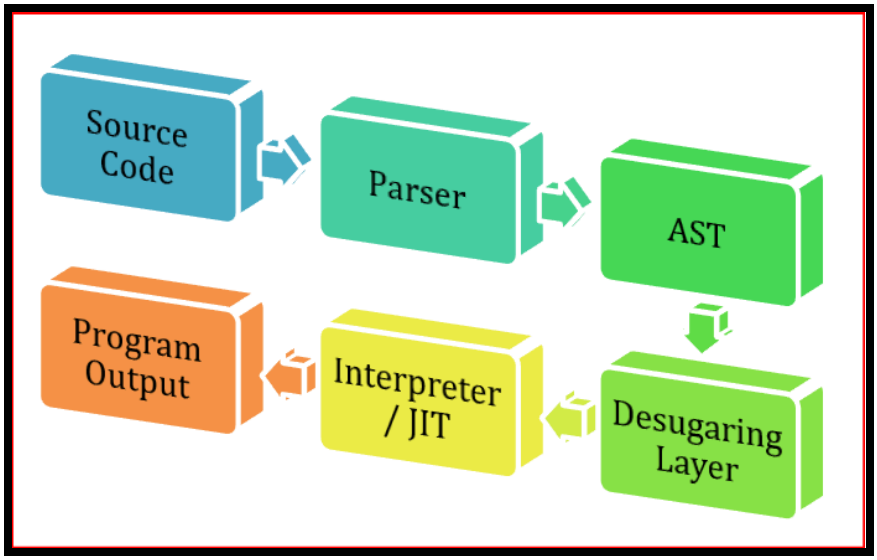
**Complier**

In the compiler, the ***curve of learning*** can be defined as slow, as in this process, the compilation needs to be done before the testing phase (Hardin *et al.,* p.8). The ***complexity*** regarding the ***debugging*** process in the compiler can be justified as high. It is also ***less flexible*** than the interpreter.

***b. Execution Model***

The proposed model can be justified as a hybrid kind of interpreter with the compilation of an optional JIT. The compilation steps of this model can be justified as in the flow, where the source code will be the first stage. After that, the parser will work, and then the implications of the abstract syntax tree (AST) will be found. Then the desugaring layer naturally takes place, which helps in the process of translating the high-level constructs into a simpler level of primitives. At this phase, execution engines take place. In this context, the AST interpreter can work as a default option, while the JIT compiler can be the optional one.

*Conceptual Diagram*




*Justification of the Metrics*

***Execution speed*** will be the critical one in this aspect, where the performance can be enhanced by 50% to 70% with the implication of JIT for the loops and functions that will be implicated in a repetitive manner. ***Error reporting*** is another critical aspect in this scenario, where the reporting is often allowed by the interpreter within only milliseconds after the input. Moreover, the ***memory usage*** of this interpreter can be justified as slightly high due to the overhead settings of JIT. However, it is absolutely acceptable for educational purposes.

**Task 2**

***a. Syntax along with Semantics Using Formal Methods***

Context-free grammar can be used in this scenario for defining the syntax alongside attribute grammars for the semantics. The example rules of a CFG can be defined as follows.

<program> → <stmt_list>

<stmt_list> → <stmt> | <stmt> <stmt_list>

<stmt> → <assign> | <print> | <if> | <while>

<assign> → <id> "=" <expr>

<print> → "print" "(" <expr> ")"

<if> → "if" <expr> "then" <stmt_list> "end"

<while> → "while" <expr> "do" <stmt_list> "end"

<expr> → <term> | <expr> "+" <term> | <expr> "-" <term>

<term> → <factor> | <term> "*" <factor>

<factor> → <number> | <id> | "(" <expr> ")"

*Semantics*

val(<expr>) = val(<term>) for simple expressions.

val(<expr1> + <expr2>) = val(expr1) + val(expr2).

Assignment: store[id] = val(expr).

print(expr) → output val(expr).

if (cond) executes <stmt_list> if val(cond) = true.

while (cond) repeatedly executes <stmt_list> until val(cond) = false.

***b. Grammar Rules, Meanings, Code Snippets, and their Semantic Analysis***

*Example Rule*

<assign> → <id> "=" <expr>

SEMANTICS: store[id] = val(expr)

*Code Snippet*

x = 15

y = 20

z = x + y

print(z)

*Semantic Analysis*

x = 15 → store[x] = 15.

y = 20 → store[y] = 20.

z = x + y → store[z] = 35.

print(z) → Output: 35.

**Task 3**

***a. Implementation of Basic Interpreter in Racket***

```
#lang racket


(define store (make-hash))


;; Expression evaluator
(define (eval-expr expr)
  (match expr
    [(list '+ a b) (+ (eval-expr a) (eval-expr b))]
    [(list '* a b) (* (eval-expr a) (eval-expr b))]
    [(? number? n) n]
    [(? symbol? s) (hash-ref store s 0)]))


;; Statement executor
(define (exec stmt)
  (match stmt
    [(list 'assign id expr)
     (hash-set! store id (eval-expr expr))]
```

```
    [(list 'print expr)
     (displayln (eval-expr expr))]
    [(list 'seq stmts)
     (for-each exec stmts)]))

;; Sample program
(define program
  '(seq ((assign x 5)
         (assign y 10)
         (assign z (+ x y))
         (print z))))

(exec program)
```

***b. Testing, Documentation, and Demonstration***

*Test 1 – Arithmetic and Print*

```
X = 2
Y = 3
Print(X*Y+9)
```

The output will be: 15.

*Test 2 – Controlled Structure*

```
x = 0
while x < 3 do
  print(x)
```

```
  x = x + 1
end
```

The output will be

0

1

2

*Challenges Faced*

In this overall process, the challenges have been met in several areas, such as mapping high-level syntax with the low-level nodes of the AST and handling the storage of the variables using the hash table of Racket. Moreover, the step of desugaring has been found a bit difficult for the recursive calls in the while loop.

*Adaption*

Simplified syntax in this programming has played a pivotal role in ensuring readability for beginners. The modularity of the interpreter has also allowed for extensions.

*Demonstration Highlights*

Immediate feedback has been given in this process by the interpreters. Support has been found for the flow of the control. The complexity has also been reduced by the implication of the desugaring process. It was also extensive for the advanced level of learners.

## References


Hardin, T., Jaume, M., Pessaux, F., & Donzeau-Gouge, V. V. (2021). *Concepts and semantics of programming languages 1: A semantical approach with OCaml and Python*. John Wiley & Sons, Incorporated.

Milewicz, R., Pirkelbauer, P., Soundararajan, P., Ahmed, H., & Skjellum, T. (2021, June). Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review. In *International Conference on High Performance Computing* (pp. 233-246). Cham: Springer International Publishing. https://www.osti.gov/servlets/purl/1884435

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, *50*(4), 911-936. https://arxiv.org/pdf/2307.07221

Wang, W. (2021, November). Empowering web applications with webassembly: Are we there yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1301-1305). IEEE. https://par.nsf.gov/servlets/purl/10312863