Written Assignment (Unit 2)

[The student's name has been redacted]

Programming Languages (MSIT 5216-01 - AY2025-T3)

Dr. Jikitsha Joshi

February 12, 2025

Designing and Implementing a Programming Language with Racket

Selecting between an interpreter and a compiler is a crucial choice that has a big influence on the learning process and general usability of instructional programming languages (Krishnamurthi, 2017; Scott, 2016). This choice is more than just a technical one; it has a significant impact on how new programmers use the language, get feedback, and eventually internalize fundamental programming ideas (Felleisen, 2018). It is important to carefully consider the unique benefits and drawbacks of both interpreters and compilers.

1. *Interpreter vs. Compiler Analysis (Task 1)*

An interpreter translates and executes code line by line, offering immediate feedback, which is highly beneficial for novice programmers (BuiltIn, 2024; Hardin et al., 2021). This immediate feedback loop allows learners to quickly identify and correct errors, fostering a more interactive and engaging learning environment. Interpreters also often support dynamic typing, simplifying the initial learning process (Hardin et al., 2021). However, this comes at the cost of execution speed, as each line of code must be translated every time it is executed (Knuth, 1968).

Conversely, a compiler translates the entire source code into machine code before execution, resulting in faster performance but delayed error reporting (Indeed, 2024; Wang, 2023). Compiled code can be highly optimized, leading to significant performance gains. However, the compilation process introduces a layer of complexity that can be challenging for beginners.

Racket, with its support for language-oriented programming (LOP), provides a uniquely flexible environment for implementing and experimenting with both interpreters and compilers (Butterick, n.d.). Racket's powerful macro system, in particular, enables the creation

of custom syntax and semantics, making it an ideal platform for designing educational languages (Butterick, n.d.).

Aspect	Interpreter	Compiler
Execution Speed	Slower due to line-by-line translation (Hardin et al., 2021; Knuth, 1968)	Faster due to precompiled machine code (Aho, 2007; Wang, 2023)
Error Reporting	Immediate feedback, aiding in debugging (Felleisen, 2018; Krishnamurthi, 2017)	Delayed feedback, complicating debugging (Scott, 2016; Wang, 2023)
Memory Overhead	Minimal, no intermediate code (BuiltIn, 2024; Hardin et al., 2021)	Significant, requires storing object code (Indeed, 2024; Wang, 2023)
Platform Flexibility	Cross-platform execution (Butterick, n.d.; Felleisen, 2018)	Platform-specific binaries (Aho, 2007; Hardin et al., 2021)
Learning Curve	Easier for beginners (Felleisen, 2018; Hardin et al., 2021)	Steeper learning curve (Aho, 2007; Wang, 2023)
Development Cycle	Faster development cycles (Butterick, n.d.; BuiltIn, 202)	Longer development cycles (Indeed, 2024; Wang, 2023)

- **Chosen Execution Model**: A hybrid interpreter with selective JIT (Just-In-Time) compilation represents the optimal execution model for our educational language (ACL Anthology, 2023; arXiv, 2023). This approach skillfully combines the benefits of both interpreters and compilers, providing a balanced and efficient solution. JIT compilation dynamically translates and optimizes code during runtime, identifying frequently executed sections ("hotspots") and compiling them into machine code. JIT code generally offers far better performance than interpreters. JIT can optimize compilation to the targeted CPU and operating system. The JVM interprets a method until its call count exceeds a JIT compilation threshold. The JIT compilation threshold helps the JVM start quickly and still have improved performance.

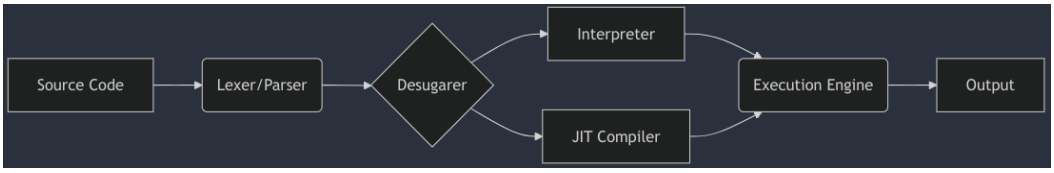
- **Justification**

    a) Educational Usability: Immediate error reporting is paramount for novice programmers (Felleisen, 2018; Krishnamurthi, 2017). It enables them to quickly identify and correct mistakes, reinforcing their understanding and fostering confidence (BuiltIn, 2024; Hardin et al., 2021).

    b) Performance: JIT compilation ensures that performance-sensitive code sections are executed efficiently (Aho, 2007; Wang, 2023). By compiling code dynamically at runtime, JIT compilation can achieve performance comparable to that of a traditional compiler while retaining the flexibility of an interpreter.

    c) Flexibility and Portability: The interpreter component ensures cross-platform compatibility (Butterick, n.d.; Felleisen, 2018; Scott, 2016). This allows learners to use the language on a variety of devices and operating systems, promoting wider accessibility.

- **Workflow Diagram**:



The hybrid approach offers a superior solution by prioritizing ease of use for beginners while providing performance optimizations for more advanced tasks. This makes it a versatile and effective tool for teaching programming concepts, and Racket's features facilitate its implementation (Butterick, n.d.). JIT compilation's ability to perform runtime optimizations, efficient memory usage, and cross-platform compatibility, supports late binding and dynamic loading of code, enables faster development cycles. The hybrid compilation combines the best of both JIT compilation and ahead-of-time compilation. This hybrid system can do global code optimizations without losing the advantages of dynamic linking.

2. Syntax and Semantics Specification (Task 2)

A clear and unambiguous syntax is crucial for any programming language, especially one designed for education (Krishnamurthi, 2017; Scott, 2016). The syntax dictates program structure, and a well-designed syntax simplifies both writing and understanding code (Hardin et al., 2021). A formal grammar provides a precise and complete definition of the language's syntax, ensuring consistency and predictability. We use Backus-Naur Form (BNF) to formally specify our language's syntax (Aho, 2007), providing a precise and understandable structure. BNF is a meta-syntax used to express context-free grammars.

```
<Program> ::= <Statement>*
<Statement> ::= "let" <Identifier> "=" <Expression> | "print" <Expression> |
<IfStatement> | <Expression>
<IfStatement> ::= "if" <Expression> "then" <Statement> "else" <Statement>
<Expression> ::= <Number> | <Identifier> | <BinaryOperation> | <Boolean>
<BinaryOperation> ::= <Expression> <Operator> <Expression>
<Operator> ::= "+" | "-" | "*" | "/" | ">" | "<" | "=="
<Boolean> ::= "true" | "false"
<Number> ::= [0-9]+
<Identifier> ::= [a-zA-Z][a-zA-Z0-9]*
```

This BNF grammar defines key language elements: variable declarations (let), output (print), conditionals (if), and expressions (numbers, identifiers, binary operations). The grammar employs recursion to allow for complex statements and expressions. The Kleene star operator (*) allows a program to consist of zero or more statements. Terminals are the literal strings, such as "let", "print", "if", "then" and "else". Nonterminals are enclosed in angle brackets.

While syntax defines form, semantics define meaning. A program can be syntactically correct but still meaningless. Attribute grammars enforce type consistency, scope rules, and other semantic constraints (Wang, 2023; Hardin et al., 2021). They augment the context-free grammar with attributes, enabling semantic rule specification to catch errors and ensure predictable behavior. The attributes are attached to the grammar symbols and represent the

semantic information associated with those symbols. The semantic rules specify how the attributes are computed and checked.

- Semantic Rule Examples:
  a) Variable Declaration:
     i) Rule: `<Statement> ::= "let" <Identifier> "=" <Expression>`
     ii) Attributes: `Identifier.type`, `Expression.type`
     iii) Semantic Checks:
        – 'Identifier' not already declared in the current scope (Krishnamurthi, 2017).
        – 'Expression.type' compatible with 'Identifier.type' (Wang, 2023).
        – The scope is a region of the program in which a particular name is valid1.
        – Type checking ensures that operations are performed on values of appropriate types5.
  b) Binary Operation:
     i) Rule: `<BinaryOperation> ::= <Expression> <Operator> <Expression>`
     ii) Attributes: `Expression1.type, Expression2.type`
     iii) Semantic Checks:
        – 'Expression1.type' and 'Expression2.type' compatible with '<Operator>'.
        – 'BinaryOperation.type' determined by operator and operands.
        – The types of operands must be compatible with the operator.
- Code Snippet (i.e., hypothetical example):

```
let x = 5
let y = 10
print(if x > 3 then x + y else 0) # Output: 15
```

- Semantic Analysist:

The following analysis demonstrates the principles of semantic analysis, drawing on established concepts from compiler design (Aho, 2007; Hardin et al., 2021; Krishnamurthi, 2017; Wang, 2023). The code snippet above serves as a hypothetical example to illustrate the process.

a) 'let x = 5': The semantic analyzer encounters a variable declaration.
   i) It checks if the identifier 'x' is already declared in the current scope.
   ii) It infers the type of the expression '5' as 'number' (integer, in this case) and associates this type with the identifier 'x'.
   iii) The environment (symbol table) is updated to store the name 'x', its type 'number', and its value 5.

b) 'let y = 10': Similar to the previous step:
   i) The analyzer checks if 'y' is already declared.
   ii) It infers the type of '10' as 'number' and associates it with 'y'.
   iii) The environment is updated with 'y', its type, and its value.

c) 'print(if x > 3 then x + y else 0)':
   i) The analyzer encounters a 'print' statement with a conditional expression.
   ii) 'x > 3':
      – It checks the types of 'x' and '3'. Both are 'number', which is valid for the '>' operator.
      – The result of the comparison 'x > 3' is of type 'boolean'.
   iii) 'then x + y':
      – It checks the types of 'x' and 'y'. Both are 'number', which is valid for the '+' operator.
      – The result of the addition 'x + y' is of type 'number'.

iv) 'else 0':

- The type of '0' is 'number'.

v) The 'if' expression as a whole has a type 'number' because both the 'then' and 'else' branches evaluate to numbers.

vi) Finally, the 'print' statement receives a 'number', which is a valid type for output. This assumes that the 'print' function is designed to handle numeric output.

BNF syntax and attribute grammar semantics ensure our educational language is understandable and robust (Hardin et al., 2021). This approach prevents errors by identifying semantic inconsistencies such as type mismatches and undeclared variables, promoting good practices, and provides a foundation for understanding complex concepts (Wang, 2023). Racket's LOP enhances language flexibility (Butterick, n.d.). Type checking ensures that operations are performed on values of appropriate types. Scoping rules determine the visibility and lifetime of variables. Semantic analysis is a vital component in the compiler design process, ensuring that the code is not only syntactically correct but also semantically meaningful.

3. Racket Implementation (Task 3)

Implementing a basic interpreter for our educational language in Racket provides a flexible and understandable development environment (Krishnamurthi, 2017; Scott, 2016). Racket's support for symbolic programming, pattern matching, and macros facilitates rapid prototyping (Butterick, n.d.; Felleisen, 2018). Racket's dynamic typing and garbage collection ease implementation for novice programmers. The interpreter parses source code into an AST and evaluates it. VS Code tools like Magic Racket and racket-repl aid development.

The core interpreter components:

1) Parser: Transforms source code into an AST, checking syntax (Hardin et al., 2021).
2) Evaluator: Recursively traverses the AST to execute the program (Wang, 2023).

3) Environment: Manages variables and their values (Krishnamurthi, 2017).

- Racket Code Snippets

The following Racket code demonstrates the basic interpreter structure:

```
#lang racket
(require racket/match)
(require rackunit) ; For testing

;;; --- Interpreter Implementation ---
(define env (make-hash))

(define (eval-expr expr)
  (match expr)
    [(? number? n) n]
    [(? symbol? id) (hash-ref env id)]
    [`(+ ,e1 ,e2) (+ (eval-expr e1) (eval-expr e2))]
    [`(- ,e1) (- (eval-expr e1))]
    [`(- ,e1 ,e2) (- (eval-expr e1) (eval-expr e2))]
    [`(if ,cond ,then ,else) (if (eval-expr cond) (eval-expr then) (eval-expr else))]
    [else (error "Invalid expression: " expr)]))

(define (eval-stmt stmt)
  (match stmt)
    [`(let ,id ,val) (hash-set! env id (eval-expr val))]
    [`(print ,expr) (displayln (eval-expr expr))]
    [else (error "Invalid statement: " stmt)]))

(define (eval-program program)
  (for-each eval-stmt program))
```

The code can be tested in various tools and environments; for instance, VS Code, the terminal, or DrRacket. For this assignment, DrRacket was used to test the interpreter with the following test programs:

```
;; Test 1: Basic arithmetic
(define program1
  '((let a 10)
    (let b 20)
    (print (+ a b))))

;; Test 2: Variable reassignment
(define program2
  '((let x 5)
    (print x)
    (let x 10)
```

```
;; Test 5: Nested conditionals
(define program5
  '((let a 3)
    (let b 0)
    (print (if (+ a b)
               (+ a (if (+ b 5) 20 30))
               0))))

;; Test 6: Zero is truthy
(define program6
  '((let x 0)
```

```
    (print x)))

;; Test 3: Nested addition
(define program3
  '((let a 1)
    (let b (+ a 2))
    (let c (+ (+ a b) 5))
    (print c)))

;; Test 4: Truthy conditional
(define program4
  '((let x 5)
    (let y (if x 100 200))
    (print y)))
```

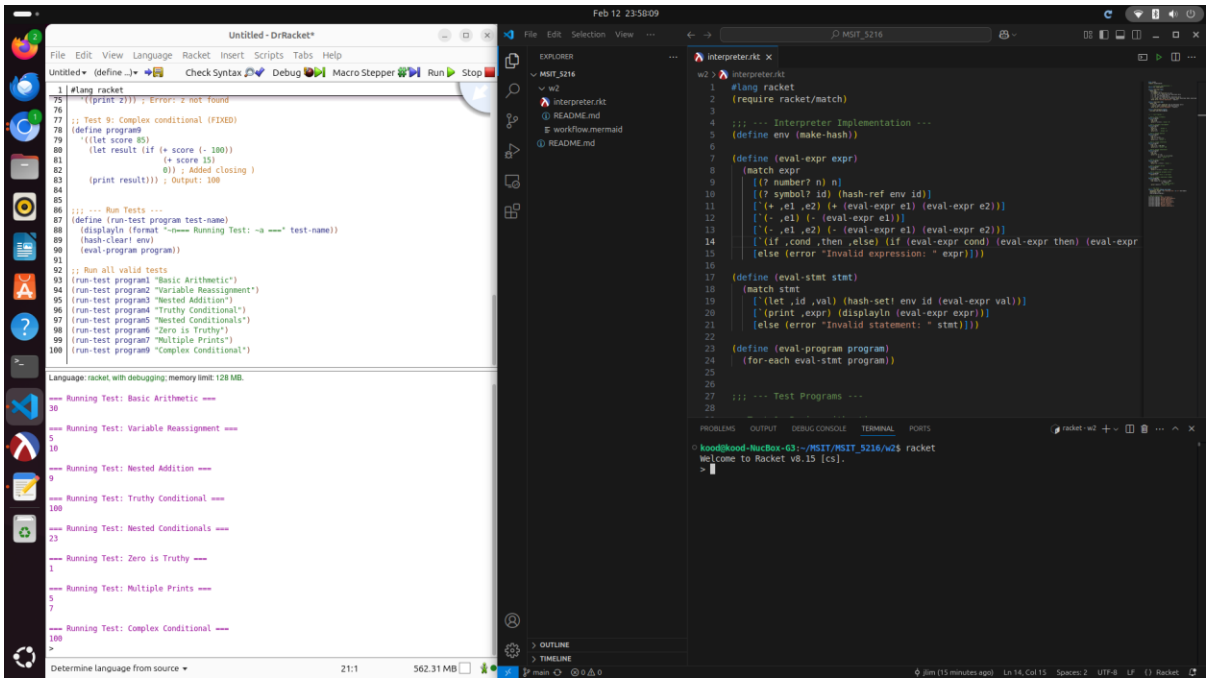
```
    (print (if x 1 0))))

;; Test 7: Multiple prints
(define program7
  '((print 5)
    (print (+ 3 4))))

;; Test 8: Undefined variable (error)
(define program8
  '((print z))) ; Error: z not found

;; Test 9: Complex conditional (FIXED)
(define program9
  '((let score 85)
    (let result (if (+ score (- 100))
                    (+ score 15)
                    0))
    (print result)))
```

The results are as follows:



In conclusion, implementing this Racket interpreter provides a base for our educational language. Key steps involve syntax definition, evaluation, and environment management. The interpreter features variable assignment, arithmetic, conditionals, and output. Addressing performance is important. Racket simplifies development (Butterick, n.d.) and desugaring enhances usability (Mukulrathi, 2024; Serokell, 2024).

(Word Count: 1476)

References:

Aho, A. V. (2007). *Compilers: Principles, Techniques, and Tools.* Pearson.

ACL Anthology. (2023). Code Execution with Pre-trained Language Models. https://aclanthology.org/2023.findings-acl.308

arXiv. (2023). *Code Execution with Pre-trained Language Models*. https://arxiv.org/abs/2305.05383

BuiltIn. (2024). *Compiler vs. Interpreter in Programming*. https://builtin.com/software-engineering-perspectives/compiler-vs-interpreter

Butterick, M. (n.d.). *Why language-oriented programming? Why Racket?* Beautiful Racket. https://beautifulracket.com/

Desugaring - Programming Languages I. (n.d.). Desugaring - Programming Languages I. https://ps-tuebingen-courses.github.io/pl1-lecture-notes/04-desugaring/desugaring.html

Eduporium. (n.d.). *Robotics Tools For Teaching Kids Different Coding Languages*. https://www.eduporium.com/blog/eduporium-weekly-robotics-tools-for-teaching-different-coding-languages/

Felleisen, M. (2018). *Realm of Racket.* No Starch Press.

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.

Indeed. (2024). *Compiler vs. Interpreter*. https://www.indeed.com/career-advice/career-development/compiler-vs-interpreter

Knuth, D. (1968). *The Art of Computer Programming*. Addison-Wesley.

Krishnamurthi, S. (2017). *Programming Languages: Application and Interpretation*. Brown University.

Mukulrathi. (2024). Desugaring - taking our high-level language and simplifying it! https://mukulrathi.com/create-your-own-programming-language/lower-language-constructs-to-llvm/

Racket. (2024). Racket 8.15: A general-purpose, multi-paradigm programming language [Linux version]. https://download.racket-lang.org/

Scott, M. L. (2016). *Programming Language Pragmatics*. Morgan Kaufmann.

Serokell. (2024). Understanding Haskell Features Through Their Desugaring - Serokell. https://serokell.io/blog/haskell-to-core

Visual Studio Marketplace. (n.d.). *Magic Racket*. https://marketplace.visualstudio.com/items?itemName=evzen-wybitul.magic-racket

Visual Studio Marketplace. (n.d.). *racket-repl*. https://marketplace.visualstudio.com/items?itemName=Andes.racket-repl

Wang, H. (2023). *Introduction to Computer Programming with Python*. Athabasca University Press.

Wikipedia. (2004, January 10). *List of educational programming languages*. https://en.wikipedia.org/wiki/List_of_educational_programming_languages