LINT: Assessing the Interpretability of Programmatic Policies

by

Zahra Bashir

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Although the synthesis of programs encoding policies often carries the promise of interpretability, systematic evaluations were never performed to assess the interpretability of these policies, likely because of the complexity of such an evaluation. In this dissertation, we introduce a novel metric that uses large-language models (LLM) to assess the interpretability of programmatic policies. For our metric, an LLM is given both a program and a description of its associated programming language. The LLM then formulates a natural language explanation of the program. This explanation is subsequently fed into a second LLM, which tries to reconstruct the program from the natural-language explanation. Our metric then measures the behavioral similarity between the reconstructed program and the original. Our evaluation is based on the literature on program obfuscation, using it as a proxy for interpretability. We validate our approach with synthesized and human-crafted programmatic policies for playing a real-time strategy game, comparing the interpretability scores of these programmatic policies to obfuscated versions of the same programs. Our LLM-based interpretability score consistently ranks less interpretable programs lower and more interpretable ones higher. These findings suggest that our metric could serve as a reliable and inexpensive tool for evaluating the interpretability of programmatic policies.

Preface

The author of this dissertation, in partnership with Michael Bowling and Levi Lelis, has produced original research. We are preparing to submit our findings for potential publication. Throughout this dissertation, the term "we" is used to acknowledge the collaborative aspect of this work. However, I take full responsibility for any technical inaccuracies or errors in presentation.

To my dad, who told me to dream big and shoot for the stars. Now, from up there among them, I hope you're smiling down on me.
And to my mom, the strongest person I've ever known!

Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less.

– Marie Curie.

Acknowledgements

I must start by extending my profound gratitude to my supervisor, Levi (Prof. Levi Lelis), for his invaluable guidance throughout my studies. Joining his group was a turning point in my academic journey, filled with endless support and opportunities for which I am deeply thankful. The most valuable lesson Levi taught me is how to think like a researcher, to approach problems critically, and to never give up early when challenges arise.

I'm really thankful for his positive attitude and the energy he shared, especially in moments of uncertainty. I'm indebted to him for always giving me the freedom to explore my interests in our research and showing great trust in me. His encouragement, regardless of the outcomes, has been a source of motivation, reminding me that the journey is as important as the destination. Developing the researcher within myself stands out as one of the most significant outcomes of working with him. To be honest, I think my path would have diverged significantly if I hadn't had the privilege of working with Levi.

I am also so grateful for the insights and advice provided by Mike (Prof. Michael Bowling) throughout this research. His guidance in refining and pitching our idea was invaluable, and his advice has significantly shaped our project.

A big thanks to Rubens Morares for his quick and clear answers to my questions on Slack. Also, an appretation to my lab mates, Zaheen, Ken, Quazi, Paul, Thirupathy, Mahdieh, David, Mahdi, Tales, Parnian, Reza, Amirhosein, Elham, Habib, and others. Our chats and brainstorming sessions enhanced our research's fun and productivity.

I really need to say a huge thank you to my wonderful mom. She's always believed in me and encouraged me to follow my heart, even though it's tough with us being so far apart. She showed me how to be a woman and be super

strong. I owe everything I've achieved to her sacrifices and wisdom throughout my life. Her love and energy propel me forward. And my lovely sister Zeinab, she's the best sister anyone could ever wish for. She's always shown pride in me, passionately engaging with my ideas and safeguarding my future and mental well-being. I'm so thankful for her constant care, and so lucky to have such a family.

I also want to honor my dad, who we lost too soon, whose wisdom and vision inspire me. I admire his problem-solving and passion for science. People say I resemble him in some ways, and I can only hope it's true. He constantly valued learning, igniting my love for research. Even though he's not here, I'm proudly his daughter, continuing on his path.

I want to give a big shoutout to my best friend, Alireza Bakhtiari, for always being there for me mentally and emotionally. He's not just a friend; he's my study buddy, gym-partner, confidant, and go-to person for meaningful conversations. Our time together is incredibly precious to me, and I'm grateful for his listening ear, and willingness to dive into ideas with me. He could easily cheer me up after a tough working day with his funny jokes. Thanks for keeping me smiling throughout my studies.

I also owe a big thanks to my friends who've been with me on this journey, especially my lovely roommates, Hasti and Kimia. They've always had my back and have been the best roomies anyone could ask for! Those long night talks filled with joy and comfort meant the world to me. I'm equally grateful to my other friends, both here and back in Iran—Fatemeh, Negin, Moein, Parham, Omid, Mina, Arad, Ehsan, Nastaran, and so many more—who've brought color into my life. Whether separated by distance or close by, their presence has made my life warmer and always been a source of happiness.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the realm of sequential decision-making problems, a policy is a function that, given a state (what the agent observes of the environment) outputs an action. Specifically, we focus on deterministic policies, where the action output by the policy is uniquely determined by the given state. In the literature, programmatic policies are deemed as policies encoded in human-readable computer programs. There is a growing interest in the use of programmatic representations of policies to solve sequential decision-making problems, both in single-agent [40], [52] and multi-agent settings [29], [30]. In addition to programmatic policies being assumed to be human-readable and thus interpretable, the interest in their use is justified, as one can provide strong inductive bias to the learning process through the domain-specific language defining the space of programs. This bias can allow programmatic policies to generalize more easily to unseen settings [20] and make them more amenable to verification [6].

Despite prior work on programmatic policies placing significant emphasis on the interpretability aspect, no systematic studies were performed that assessed the actual interpretability of these policies. A common method is to present specific programs and claim their interpretability [2], [52]. For example, Figure 1.1 illustrates a programmatic policy for controlling a car in a racing domain. This program was automatically generated by the Neurally Directed Program Search (NDPS) algorithm [52]. In the work, this programmatic policy is considered interpretable and the authors provide an explanation of it. The same anecdotal approach is taken in other works (e.g., [2]). We are

interested in having a tool that is able to automatically and systematically evaluate the interpretability of such programmatic policies.

$$
\begin{aligned}
&\textbf{if } (0.001 - \textbf{peek}(hTrackPos, -1) > 0) \textbf{ and } (0.001 + \textbf{peek}(hTrackPos, -1) > 0) \\
&\textbf{then} \\
&\quad 3.97 \cdot \textbf{peek}(0.44 - hRPM, -1) \\
&\quad + 0.01 \cdot \textbf{fold}(+, 0.44 - hRPM) \\
&\quad + 48.79 \cdot (\textbf{peek}(hRPM, -2) - \textbf{peek}(hRPM, -1)) \\
&\textbf{else} \\
&\quad 3.97 \cdot \textbf{peek}(0.40 - hRPM, -1) \\
&\quad + 0.01 \cdot \textbf{fold}(+, 0.40 - hRPM) \\
&\quad + 48.79 \cdot (\textbf{peek}(hRPM, -2) - \textbf{peek}(hRPM, -1))
\end{aligned}
$$

Figure 1.1: An example of a programmatic policy that is assumed to be interpretable. The function $\textbf{peek}(x, i)$ returns the observation from the i-th time step from history x, where $\textbf{peek}(x, -1)$ represents the most recent observation. The function $\textbf{fold}(f, [e_1, ..., e_k], e) = f(e_k, f(e_{k-1}, \dots f(e_1, e)))$ is a higher-order combinator. These functions are used to simulate components of Proportional–integral–derivative (PID) controllers.

The scarcity of comprehensive evaluations could be attributed to the fact that such studies are time consuming and costly, mainly because they would involve human programmers. This lack of a thorough analysis hinders our understanding of what precisely makes a programmatic policy interpretable. For instance, neural networks can be viewed as programs written in a domain-specific language that allows the addition of layers and nodes to the neural architecture—clearly, the programmatic framing for policies does not guarantee interpretability. So, what are the properties that make a programmatic policy interpretable?

Any viable approach to addressing this question is likely to involve evaluating the interpretability of programmatic policies. In this dissertation, we introduce a simple and cost-effective methodology to assess program interpretability and demonstrate its application to programmatic policies. Our assumption is that if a program is interpretable, it can be reconstructed from a natural language description. In other words, we use a natural language bottleneck to verify the interpretability of the program. If we can successfully

reconstruct the program later, we consider it to be interpretable.

This natural language bottleneck is achieved with Large Language Models (LLMs) [9]. Our methodology uses LLMs to assign an interpretability score to a program. We call this score the **L**LM-based **INT**erpretability (LINT) score. In our methodology, we use an instance of an LLM to generate a natural-language explanation of a program. This explanation is given as input to another instance of an LLM, which is asked to reconstruct the program described in the explanation. A third instance of an LLM verifies that the explanation is in natural language and does not provide step-by-step programming instructions on how to write the program. The LINT score is the value of a metric comparing the behavior of the original and reconstructed programs. We introduce general behavior metrics for sequential decision-making problems.

The evaluation of our methodology is based on methods from the program obfuscation literature [12]. Obfuscated programs are designed to be non-interpretable, and some obfuscation techniques allow us to construct programs with different levels of obfuscation. Assuming that obfuscation can be used as a proxy for interpretability, we hypothesize that the LINT scores negatively correlate with the degree of obfuscation we apply to the programs. Our methodology also includes the use of programmatic policies written by humans. Our premise is that since these policies are human-written, they should be inherently interpretable, and thus be scored as such in our metric. This assumption is based on the fact that the programs used in our study were written with the goal of being interpretable by others.

We test this hypothesis in two domains: classical programming problems and programmatic policies for playing MicroRTS [35], a real-time strategy game. Empirical results on these two domains show that the LINT scores negatively correlate with the level of obfuscation of the programs evaluated. Although user studies should still be the gold standard for evaluating interpretability, our results suggest that LINT can be used as a reliable and inexpensive tool to help drive research in interpretable programmatic policies.

In conclusion, this dissertation evaluates the interpretability of programmatic policies based on the hypothesis that if a program can be reconstructed

from a natural language description generated by an LLM—a natural language bottleneck—then it would be considered interpretable to people.

This dissertation is organized as follows: Chapter 2 provides explanations of key concepts such as program interpretability, code understandability, large language models, code obfuscation, and program synthesis, which are essential for understanding the subsequent discussions. Chapter 3 introduces the LINT score, a novel metric for assessing the interpretability of programmatic policies using large language models. Chapter 4 describes the empirical methodology, detailing the procedures and introducing domains used for testing the interpretability metric. Chapter 5 presents the results of the empirical tests, analyzing the effectiveness of the LINT score in various settings. Chapter 6 discusses the implications of the findings, potential limitations of the study, and future research directions. Chapter 7 concludes the dissertation by summarizing our contributions in the context of programmatic policies and their interpretability. The dissertation also includes a section on supplementary materials (Chapters A and B), featuring an introduction to the domain used in this dissertation, the prompts, examples, and set of programs from our experiments.

Chapter 2

Background

In this chapter, we delve into the subjects of Program Interpertability, Code Understandability, Large Language Models, Code Obfuscation, and Program Synthesis, which constitute the essential background needed to navigate through this dissertation.

2.1 Program Interpertability

Defining interpretability (also referred to as "explainability") within the context of mathematical frameworks presents a significant challenge. A compelling non-mathematical definition proposed by Miller [32] offers a valuable perspective: "Interpretability is the degree to which a human can comprehend the cause of a decision."

In the context of programs, a program is deemed interpretable when individuals can fully comprehend its functionality and the manner in which it interacts with its operational environment. This comprehensive understanding enables users to accurately anticipate the program's behavior and the outcomes of its execution under various conditions.

The contrast between interpretable and non-interpretable programs is illustrated through the examples provided below. Figure 2.1, written in C language, is designed to compute a proper subset of a set of arguments passed to it. However, its complexity and the manner in which it is coded make it a non-interpretable program. This program is also called an "obfuscated program" in the Software Engineering literature [12]. More details on "Code Obfuscation"

can be found in Section 2.4. In contrast, Figure 2.2, written in C language as well, presents an equivalent program that maintains the same functionality but is written in a manner that greatly enhances its interpretability.

```
main(Q,O)char**O;{if(--Q){main(Q,O);O[Q]
 [0]^=0X80;for(O[0][0]=0;O[++O[0][0]]!=0;)
 if(O[O[0][0]][0]>0)puts(O[O[0][0]]);
 puts("----------");main(Q,O);}}
```

Figure 2.1: A non-interpretable program that computes all the proper subset of the set of arguments passed to it.

Another related concept is **"Explainability"**. Drawing from the work of Kim et al.[23], we define an explainable model as a system capable of providing explanations by documenting its prediction process. For instance, in a rule-based system, this would involve detailing the sequence of rules activated by a given input. Explainability focuses on the process of providing detailed justifications for an inference or prediction made by a model, regardless of whether the model is easily interpretable or only loosely interpretable. In terms of explainability, our interest lies in explaining the decisions made by the system. However, interpretability concerns our understanding of the model itself. Within the scope of our research, we treat a program as a model.

```
void subsets(char *av[], int c, int n,
char *sbset[], int sz) {
    if (c == n) {
        if (sz < n) {
            for (int i = 0; i < sz; i++)
                printf(sbset[i]);
            printf("----------");
        }
        return;
    }
    sbsets(av, c+1, n, sbset, sz);
    sbset[sz] = av[c];
    sbsets(av, c+1, n, sbset, sz+1);}
```

Figure 2.2: An interpretable program that computes all the proper subset of the set of arguments passed to it.

2.2 Code Understandability

Code understandability has been extensively studied within the Software Engineering research community. Studies such as those by [10], [13], [34], [39], [45] explore metrics for assessing code understandability, tackling a problem closely related to program interpretability, which is the focus of this dissertation. These issues can be considered analogous, though they involve different inputs and contexts: code snippets within the realm of software engineering versus programmatic policies in this study. Code understandability refers to the ease with which a developer can comprehend a codebase, crucial for effectively fixing bugs or adding new features in a timely manner. A thorough understanding of code helps estimate the effort required to modify code components and guides developers in writing better, more maintainable code. Unfortunately, despite extensive research, there are currently no definitive metrics[44] designed specifically to assess the understandability of code snippets, leaving a gap in quantifiable standards for code clarity and readability. This dissertation introduces a metric aimed at bridging this gap and providing a unified way of measuring the interpretability of programs.

2.3 Large Language Models

Large language models (LLMs) are a class of artificial neural network optimized for natural language tasks, renowned for generating text that closely mimics human writing. These models, which are large in scale, employ transformer-based architectures as introduced by Vaswani et al. [51] in 2017. This architecture is pivotal for learning intricate data patterns, setting the stage for breakthroughs like BERT [14] and GPT [41]. Over time, models have advanced significantly, with GPT-3 [8] emerging as a landmark for its extensive scale and proficiency in various tasks without the need for specific training.

LLMs are used in a wide variety of tasks, including writing text, translating languages, summarizing content, and generating computer code. Their flexible and all-purpose nature makes them essential tools in advancing artificial

intelligence's ability to understand and interact with human language.

2.3.1 Prompt Engineering

2.3.1.1 What is a prompt?

A prompt is a set of instructions given in natural language to a generative AI system, enabling it to create content such as stories [18], images [42], [43], or music [16]. These instructions guide AI models, trained on extensive datasets, to generate outputs based on the context provided. Although generative AI models, especially large language models, can start creating from minimal inputs, not all prompts yield useful results. Effective prompt engineering involves crafting detailed prompts to produce precise and relevant responses, refining these prompts to achieve desired outcomes.

2.3.1.2 What is Prompt Engineering?

Prompt engineering involves guiding AI systems designed to generate content (known as generative AI) towards producing specific results. Although these systems aim to replicate human-like output, they need clear instructions to produce relevant and high-quality content. This process involves selecting the best phrases, words, and symbols to help the model better engage with users. Through a mix of creativity and experimentation, prompt engineers develop a set of inputs that ensure the AI system operates as intended, enhancing its interaction with applications [8].

Consider the task of generating a Python function to calculate the factorial of a number. The initial prompt and its output are illustrated in Figure 2.3. Next, some refinement through prompt engineering is applied, which can significantly affect the quality and specificity of the generated code illustrated in Figure 2.4. The progression from a basic to a refined prompt demonstrates the essence of prompt engineering. By adjusting the prompt to specify an"efficient" and "iterative" approach, we guide the AI system to generate code that is more suited to our requirements, illustrating the practical application of prompt engineering principles.

```python
# Prompt: "Write a Python function to calculate factorial."
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Figure 2.3: A basic prompt and its generated output.

```python
# Prompt: "Write an efficient Python function to calculate
   factorial using an iterative approach to avoid stack
   overflow with large inputs."
def factorial_iterative(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

Figure 2.4: A refined version of the prompt presented in Figure 2.3 and its corresponding output.

2.3.1.3 Chain of Thought Prompting

Chain-of-thought prompting is a method in prompt engineering where a complex query is divided into smaller, logical steps, mirroring human thought processes. This technique allows the model to navigate through a problem progressively, enhancing its reasoning capabilities. For instance, to determine "What is the capital of Canada", the model might explore various logical paths, eventually leading to conclusions such as "Ottawa," "The capital of Canada is Ottawa," and "Ottawa is the capital of Canada." Given that all paths arrive at the same endpoint, "Ottawa" would be affirmed as the correct answer [53].

Illustrative Example: Calculating the Area of a Painted Wall

Consider you want to calculate the area of a wall that needs painting, but the wall has a door that will not be painted. The wall's dimensions are 10 feet wide by 8 feet high, and the door measures 3 feet by 7 feet. A direct prompt might ask, "What is the area of the wall that needs painting?" However, us-

ing chain of thought prompting, the query is broken down into smaller, logical steps that lead to the final answer.

Prompt powered by Chain of Thought Prompting:

1. Calculate the area of the wall.
2. Calculate the area of the door.
3. Subtract the area of the door from the area of the wall to determine the painting area.

AI Model's Response:

```
1. The area of the wall is 10 feet * 8 feet = 80 square feet.
2. The area of the door is 3 feet * 7 feet = 21 square feet.
3. The painting area is 80 square feet - 21 square feet = 59 square feet.
```

Thus, the area that needs painting is calculated to be 59 square feet, demonstrating how chain of thought prompting can effectively guide the model through a series of reasoning steps to arrive at the correct answer.

2.4 Code Obfuscation

The empirical methodology of our study is grounded in the literature on code obfuscation. This involves employing obfuscation as a means to assess interpretability, underscoring its significance in our research framework. Given this, we now proceed to define code obfuscation itself. Code obfuscation is a technique used to make software code more difficult to understand or interpret without affecting its functionality. The primary goal of code obfuscation is to protect the code from unauthorized access, reverse engineering, or tampering, thereby securing intellectual property or sensitive data embedded within the code. Obfuscation is widely used in software development, especially for applications that are distributed or deployed in insecure environments [12]. Figure 2.1 represents an obfuscated C program.

2.4.1 Code Obfuscation Techniques

Here are some common techniques used for code obfuscation [12]:

- Name Obfuscation: This involves changing the names of variables, classes, and methods to meaningless or misleading names. It makes it harder for someone to understand the code's purpose or functionality just by reading it.
- Control Flow Obfuscation: This technique alters the execution path of the program without changing its output. It introduces conditionals, loops, and other control statements to make the flow of the program difficult to trace.
- String Encryption: Sensitive information within the code, such as database passwords or API keys, is encrypted. The code then decrypts this information at runtime. This prevents easy extraction of sensitive data from the code.
- Instruction Pattern Transformation: The code is transformed into an equivalent but less obvious form. For example, simple operations might be replaced by more complex, equivalent expressions, making the code harder to analyze.
- Dead Code Insertion: Also known as "junk code insertion," this involves adding code that does not affect the program's functionality. Its purpose is to confuse anyone trying to reverse engineer or understand the code's purpose.
- Dynamic Obfuscation: This involves techniques that modify the code at runtime. For example, a program might decrypt, execute, and then re-encrypt a portion of its code while running, making static analysis extremely difficult.
- Opaque Predicates: These are conditions added to the code that always evaluate to true or false but in a way that is not obvious. This can significantly complicate the control flow analysis.

- Code Virtualization: This technique involves converting portions of the code into a virtual instruction set that is interpreted by a custom virtual machine (VM) at runtime. This adds an additional layer of abstraction, making the code harder to analyze directly.

- Anti-debugging Techniques: These are methods used to make debugging the obfuscated code more difficult, either by detecting and interfering with debuggers or by making the code behave differently under a debugger.

Illustrative Examples

Below, you can see a list of codes and their obfuscated version with the technique used for their obfuscation:

Example 1: Dead Code Insertion

```c
int Factorial(int n) {
    if (n <= 1) return 1;
    else return n * Factorial(n - 1);
}
```

Original factorial function in C.

```c
int Factorial(int n) {
    int result = 1;
    if (n > 1) {
        result = n * Factorial(n - 1);
    }
    int deadCode = result - result; // Dead code
    return result + deadCode;
}
```

Obfuscated factorial function with Dead Code Insertion.

Figure 2.5: Comparison between Original and Obfuscated Code using Dead Code Insertion.

Example 2: Name Obfuscation

```c
int calculateTotalExpenses(int prices[], int count) {
    int total = 0;
    for(int i = 0; i < count; i++) {
        total += prices[i];
    }
    return total;
}
```

Original code in C

```c
int x82y(int a[], int b) {
    int c = 0;
    for(int d = 0; d < b; d++) {
        c += a[d];
    }
    return c;
}
```

Obfuscated version of the original code with Name Obfuscation.

Figure 2.6: Comparison between Original and Obfuscated Code using Name Obfuscation.

Example 3: x Obfuscation [26]

```c
while(1){
    break;
}
```

Original code in C.

```c
int swVar = 1;
while(swVar != 0){
    switch(swVar){
        case 1:{
            if (1)
                swVar= 2;
            else
                swVar= 0;
            break;
            }
        case 2:{
            swVar= 0;
            break;
            }
        }
    }
```

Equivalent logic implemented with a switch statement.

Figure 2.7: Comparison between Original and Obfuscated Code using Control Flow Obfuscation.

2.5 Program Synthesis

Program synthesis is an active area of research in Artificial Intelligence and in Programming Languages. According to Solar-Lezama [47], it corresponds to a class of techniques that are able to generate a program from a collection of artifacts that establish semantic and syntactic requirements for the generated code. In other words, it can be considered as a search in the program space that satisfy a specification λ.

In the context of program synthesis, λ typically represents a formal specification that the synthesized program must satisfy. This specification can take various forms, including but not limited to logical expressions, input-output examples, or higher-level descriptions of behavior. For instance, to synthesize a function that identifies the maximum number in a list, λ could be specified through input-output pairs like: for input `[1, 5, 4]` the output should be 5, for `[-2, -7, -3]` it should be -2, and for `[6]` it should be 6.

The program space is defined by a Domain-Specific Language (DSL) [50]. There are different types of search algorithms for solving program synthesis problems such as Bottom-Up Search [1], [3], [49], Top-Down Search [4], [27], and Stochastic Local Search [19], [24].

A simplified grammar of a programming language is presented in Figure 2.8, expressed in Backus-Naur Form (BNF). This grammar defines the syntactic structure of a programming language, where a program consists of statements that can either assign values to variables or execute conditional logic through "IfThenElse" constructs. Expressions within this language allow for basic arithmetic operations (addition) and comparisons. Variables are represented by single lowercase letters, and integers are defined as non-negative numbers.

Using this grammar, a program synthesizer can generate code snippets that follow the given syntactic rules. The synthesizer searches in the space of possible programs that meet a specific condition or goal, as defined by a λ expression. This process involves exploring various combinations of statements, expressions, and control structures until the generated program aligns with the λ's requirements. An example result based on the grammar, is shown in

```
Program ::= Statement
Statement ::= Assignment | IfThenElse
Assignment ::= Variable = Expression;
IfThenElse ::= if ( Condition ) { Assignment }
else { Assignment }
Expression ::= Term | Expression + Term
Term ::= Integer | Variable
Condition ::= Expression == Expression |
Expression < Expression | Expression > Expression
Variable ::= x | y | z | ...    (single lowercase letters)
Integer ::= 0 | 1 | 2 | ...    (non-negative integers)
```

Figure 2.8: Simplified Grammar of a Language in BNF Notation.

Figure 2.9. The synthesized program demonstrates how variables x and y are assigned integer values, followed by a conditional statement that evaluates the sum of these variables. Depending on the result of the conditional evaluation, variable z is assigned a corresponding integer value.

```
x = 5;
y = 7;
if (x + y > 10) {
    z = 1;
} else {
    z = 0;
}
```

Figure 2.9: An example of a synthesized program using the grammar in Figure 2.8.

2.5.1 Synthesizing Programmatic Policies

Initially, it is crucial to comprehend the concept of a policy. A policy is defined as a function that receives a game's state as input and outputs the action an agent should execute in that state. Programmatic policies are policies encoded in human-readable computer programs. This type of programmatic represen-

tation enables a deeper understanding of, as well as the ability to modify, the encoded policy [33]. Figure 2.10 presents an example of a programmatic policy designed for playing the board game "Can't Stop". This policy includes elements (Lines 4 and 15) that have been synthesized based on the methodology proposed by Medeiros et al. [31]. These lines were synthesized through a technique known as "sketch-learning" [46], a program synthesis search algorithm. The framework of the code builds upon the foundational strategy developed by Glen and Aloi [17], which was considered the most effective programmatic approach until the publication of Medeiros et al.'s paper [31]. The enhancements introduced by the newly synthesized code successfully outperform previous programmatic solutions, marking a significant advancement in strategic game play for Can't Stop. This showcases the application of program synthesis techniques in game strategy development. Another example of a synthesized programmatic policy can be found in Figure A.2.

```
1.  def get_action(self, state):
2.      actions = state.available_moves()
3.      if actions == ['y', 'n']:
4.          score = sum(map(lambda x: (f1 + 1) * l4, l2)) + f5
5.          if win_after_n(state):
6.              return 'n'
7.          elif available_columns(state):
8.              return 'y'
9.          else:
10.             if score >= 29:
11.                 return 'n'
12.             else:
13.                 return 'y'
14.     else:
15.         index = argmax(map(lambda x:\
                sum(map(lambda x: f2 * l5 - 6 * f6, l1)), l3))
16.         return actions[index]
```

Figure 2.10: Decision-making policy for playing "Can't Stop", with codes on lines 4 and 15 synthesized via program synthesis techniques to optimize moves.

Next, we will explore the framework and terminology of two-player zero-sum games, including critical concepts such as "strategy" and "best response,"

to better grasp the foundational language of the field. The algorithms and concepts that we will cover in the next section can be used to synthesize programs similar to the one shown in Figure 2.10.

2.5.2 A Two-Player Zero-sum Game Setting

Sequential two-player zero-sum games are defined by a set S of states, a pair of players $\{i, -i\}$, an initial state s_{init} within S, a function $A_i(s)$ that, given a state s, returns the set of actions available for player i at s, and a function $U_i(s)$ that provides the utility of player i at s. In the context of zero-sum games, $U_i(s) = -U_{-i}(s)$. The policy for player i is represented by a function $\sigma_i : S \to A_i$, which maps a state s to an action a.

A programmatic policy encapsulates a policy σ within a computer program. The value of the game for player i for a state s, assuming players i and $-i$ adhere to policies σ_i and σ_{-i} respectively, is denoted as $U_i(s, \sigma_i, \sigma_{-i})$. That is, this is assuming that starting from state s, player i will follow the policy σ_i and player $-i$ will follow the policy σ_{-i} until the end of the game.

The domain-specific language (DSL) [50] is utilized to describe the program space, thereby defining the policy space for game play. D signifies a DSL, with $[\![D]\!]$ representing the (potentially infinite) collection of programs that can be written using D.

The best response of σ_{-i} in $[[D]]$ maximizes player i's utility against σ_{-i}, denoted as

$$\max_{\sigma_i \in [[D]]} U(s_{\text{init}}, \sigma_i, \sigma_{-i}). \tag{2.1}$$

In two-player zero-sum games, a Nash equilibrium profile within the domain of programmatic strategies consists of a pair of programs that mutually constitute the best response, specifically, strategies σ_i and σ_{-i} satisfying

$$\max_{\sigma_i \in [[D]]} \min_{\sigma_{-i} \in [[D]]} U(s_{\text{init}}, \sigma_i, \sigma_{-i}). \tag{2.2}$$

The objective is to approximate a solution for Equation 2.2. Given that the DSLs we consider in our work encode pure strategies, the existence of a pure-strategy Nash equilibrium is assumed.

Consider the following example. Poachers & Rangers (P&R) is a game played with simultaneous moves (illustrated in Figure 2.11), featuring two participants: one as poachers and the other as rangers, in a setting that is strictly competitive with no possibility for ties. The game's essence revolves around the rangers' mission to protect the entry points of a national park from poachers. Victory is secured by the rangers if they successfully defend all targeted gates, earning them a score of +1; conversely, failure to protect any gate from a poaching attempt results in a loss and a score of -1. While at first glance, the rangers' optimal strategy may appear straightforward—guard every entrance—the challenge intensifies when this strategy is viewed as a task of program synthesis, especially as the number of gates is large. Existing program synthesizers may find it difficult to construct programs for a game with a large number of gates. For instance, in a scenario with n gates, the ideal programmed strategy would consist of a sequence of defensive commands: `defend[1]`, `defend[2]`, ..., `defend[n]`. This sequence is concisely denoted as `defend[1, 2, ..., n]` for simplicity. This strategy is a Nash Equilibrium profile [33].

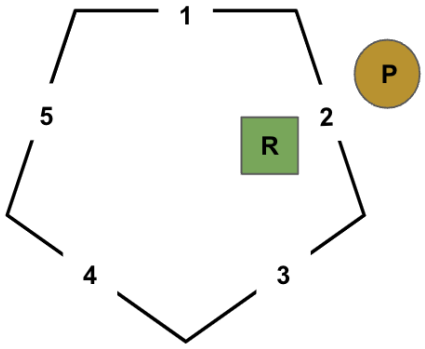

Figure 2.11: A illustration of P&R game with 5 gates. In this specific instance, the poacher launches an attack on gate 2, which is simultaneously defended by the ranger.

2.5.2.1 Self-Play Algorithms

The synthesis of programmatic strategies requires one to search in large non-differentiable spaces of computer programs. Current search methods use self-

play algorithms to guide this search. We briefly describe three self-play algorithms.

2.5.2.1.1 Iterated Best Response(IBR) In a game scenario, in IBR, one starts with an arbitrary strategy σ_i for player i and computes a best response σ_{-i} to σ_i. Then, in the next iteration of IBR, it computes a best response to σ_{-i}, and so on. This cycle of best responses continues for several iterations, and the strategy developed in the final iteration is returned as an approximate solution to Equation 2.2 [2], [33]. Consider an instance of a Poachers & Rangers with 2 gates, below you can see how IBR works for this instance of the game.

Iteration	Poachers' Response	Rangers' Strategy	Note
1	-	`defend[2]`	Initial arbitrary strategy by Rangers
2	`attack[1]`	`defend[1]`	-
3	`attack[2]`	`defend[2]`	Best response to attack[2], loops back

Table 2.1: Iterated Best Response (IBR) Procedure in P&R Game.

Note that IBR can loop through a series of suboptimal strategies due to its reliance on the last synthesized strategy, potentially delaying convergence to the optimal strategy of defending all gates (`defend[1, 2]`).

2.5.2.1.2 FP Fictitious Play (FP) [7], like Iterated Best Response (IBR) [25], begins with a random starting strategy σ_i for each player and calculates the best response to it. Unlike IBR, FP keeps track of two sets of strategies for each player: Σ_i and Σ_{-i}, with all the best responses computed for each player. In each round of FP, a best response is calculated against a mixed strategy that includes all the strategies in Σ_i (or Σ_{-i}). Over time, the mix of strategies in these collections evolves towards a mixed-strategy Nash equilibrium for the game. When focusing on purely programmed strategies, FP is run until a set time limit is reached. At this point, the last best response noted for each set

is used as the algorithm's estimate of a pure-strategy Nash equilibrium [2]. In Table 2.2, you can see how FP works for an instance of P&R initiated with a random strategy by the Rangers.

Iteration	Cumulative Poachers' Responses	Cumulative Rangers' Strategies
1	-	`defend[1]`
2	`attack[1]`	`defend[1]`
3	`attack[1, 2]`	`defend[1, 2]`
4	`attack[1, 2, 3]`	`defend[1, 2, 3]`

Table 2.2: Fictitious Play (FP) Procedure in P&R Game.

Note that FP considers all previously synthesized strategies during the learning process. Once it encounters the strategies to attack three gates (attack[1, 2, 3]), the algorithm is guided to synthesize the optimal strategy of defending all gates (`defend[1, 2, 3]`).

2.5.2.1.3 Local Learner (2L) 2L is a variant of FP designed to guide algorithms searching for programmatic strategies. It leverages data from computing best responses to select target strategies for future algorithm iterations, optimizing the search signal. 2L establishes meta-strategies that bridge the gap between IBR and FP by varying the number of strategies in the meta-strategy's support [25]. IBR's meta-strategy only includes the last best response the algorithm computed; FP's meta-strategy includes all best responses computed. 2L utilizes more strategies than IBR for improved search signals, while aiming to use fewer than FP to lower computational costs. 2L initially assumes that all the strategies inserted in the meta-strategy are helpful in guiding the search. While computing a best response to σ_{-i}, it collects data on each strategy in σ_{-i} and removes from the support of the meta-strategies all redundant best responses [33]. In Table 2.3, you can see how 2L works for the an instance of P&R with $n > 2$ gates initiated with a random strategy by the Rangers.

Iteration	Meta-Strategy for Rangers (σ_i)	Meta-Strategy for Poachers (σ_{-i})
1	-	`attack[2]`
2	`defend[2]`	`attack[1]`
3	`defend[2]`	`attack[1, 2]`
4	`defend[2]`, `defend[1, 2]`	-
4	`defend[1, 2]` (`defend[2]` is dropped)	...

Table 2.3: Local Learner (2L) Iterative Process in P&R Game with $n > 2$ Gates.

The iterative process in Table 2.3 initiates with an arbitrary starting strategy for the Poachers, with the Rangers' best response being to `defend[2]`. As the iterations advance, both Rangers and Poachers adjust their meta-strategies in response to the outcomes of previous rounds. This adaptation leads to the inclusion of all possible strategies within the meta-strategies for both parties, which facilitates the identification and elimination of redundant strategies. Note that in this example, `defend[2]` is a best response to only `attack[2]`, while `defend[1, 2]` is a best response to both. So, `defend[2]` does not add new information to the search and can be dropped.

Before delving into the chapters of the dissertation, it is important to clarify that we will use the term "policy" instead of "strategy" throughout the text. This is because the process of computing best responses can be framed as a single agent problem, in which the opponent is considered a part of the environment.

Chapter 3

LINT: LLM-based Interpretability Score

We define the function $B(\pi_1, \pi_2)$ as a similarity metric for the behavior of two programs. We consider functions B that return a number between 0 and 1, where the value of 0 represents the most dissimilar behavior for the two programs and 1 represents identical behavior for the programs.[1] We denote by $\mathcal{L}_e(\pi, G, C)$ an LLM that receives a program π, a domain-specific language (DSL) G, a set of constraints C, and returns a natural language explanation of π. We refer to this LLM as **explainer**. We denote by $\mathcal{L}_r(e, G)$ an LLM that receives a natural language explanation e of a program and a DSL G, and returns a program accepted by the language G that exhibits the behavior described in e. We refer to this LLM as the **reconstructor**. Both the explainer and the reconstructor receive a natural language description of G with a context-free grammar that specifies the programs G accepts.

Given a set Π with n programs, a behavior metric B, a grammar G describing the DSL in which the programs n are written, and a set of constraints C, the LINT score is computed as

$$\text{LINT}(\Pi, B, G, C) = \frac{1}{n} \sum_{\pi \in \Pi} B(\pi, \mathcal{L}_r(\mathcal{L}_e(\pi, G, C), G)) \,. \tag{3.1}$$

The LINT score of set Π is the average value of how similar the programs in Π are from the reconstructed ones. We define the LINT score over a set of

[1] In our experiments, we also consider a dissimilarity metric, where 0 represents the most similar and 1 the most dissimilar behavior.

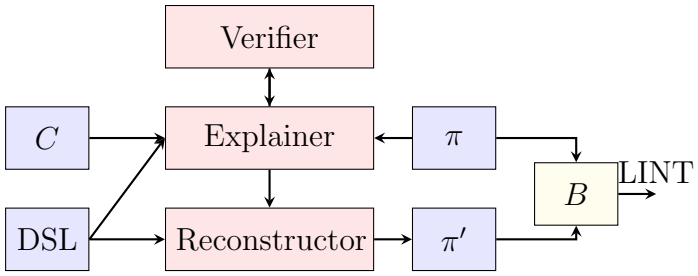

Figure 3.1: General overview of LINT. The Explainer receives a program π, a set of constraints C, and a description of the DSL in which π was written; it produces a natural language explanation of π, which is checked by the Verifier. The explanation is provided as input, along with the description of the DSL, to the Reconstructor, which attempts to reconstruct π from the explanation, thus producing π'. B scores the similarity (or dissimilarity) of π and π'.

programs to measure the interpretability of the programs a system generates. However, in our experiments we also consider the case where $|\Pi| = 1$.

Figure 3.1 shows a schematic view of how the LINT score is computed for a program π. A suitable definition of Program Interpretability for the purposes of this research could be articulated as: **"Explainability with the goal of Replicability"**. In other words, if a program can be conveyed through the natural language bottleneck of our system, explained clearly, and subsequently reconstructed, we hypothesize that such a program is interpretable.

3.1 Set of Constraints for Explanation

The above formulation considers a set of constraints C to generate the explanation of a program. C prevents the LLM from generating the explanation of the program with non-interpretable elements that communicate the program to the other LLM. The constraints are instructions in the LLM prompt. We include the constraints shown in the list below.

1. *Try to understand what is happening in the code and explain it in natural language to someone who wants to learn about this program.*

2. *Write a high-level explanation and do not explain the code line-by-line, but it is fine to include numbers in your natural language explanation.*

3. *You must not use programming language jargon as people not familiar with programming might not understand the explanation.*

Without these constraints, the LLM could generate line-by-line instructions of how to reconstruct the program. For example, even if the program was an implementation of the neural network, the LLM could provide instructions on how to implement the architecture and copy the weights of the model. Although this explanation could allow the second LLM to reconstruct the program, the original program might not be interpretable. Even with these constraints, the LLM occasionally generates explanations that use programming instructions such as *"[...] after a nested for-loop [...]"*. We use a third LLM, the **verifier**, to check for the constraints. Specifically, we ask it to verify whether the explanation uses computer programming jargon and/or keywords of the DSL. If the verifier answers 'yes' to the use of jargons, then we sample another explanation from the explainer.

There is a concern about using an LLM as a "Verifier" in our system. The issue comes from understanding that even the best language models might not always get instructions right, which is why we thought we needed a Verifier in the first place. The main job of this Verifier is to check the program for any programming elements, jargons or hints that we do not want it to be there. So, it stands to reason that the Verifier also cannot be trusted to perform its job consistently. The concern raised is acknowledged, and to address it, we propose a *hierarchical verification strategy*. Essentially, we can wrap our Verifier within multiple layers of verification, creating a Verifier for the Verifier, and so on. By implementing numerous levels of evaluation, we aim to significantly reduce the probability of error through this layered approach.

3.2 Multiple Trials

Due to the stochastic nature of how the LLMs generate the explanations and programs, we repeat k times the computation of $B(\pi, \mathcal{L}_r(\mathcal{L}_e(\pi, G, C), G))$ in Equation 3.1 and use in the summation the best B-value of the k trials. Trials are carried out by generating one explanation for each program, and each

explanation is used to generate k programs. The value of k should be large enough to account for the variance of the LLM generation and small enough to prevent the LLM from reconstructing the original program by chance. Since the program space is vast, as we evaluate empirically, it is safe to use a few trials to compute the LINT score without allowing the LLM to reconstruct the correct program by chance. For more information on why we need k trials, please refer to Section 6.1.2 on hallucination.

3.3 Caveats of LINT Score

When assessing the interpretability of programs, we assume a level of knowledge of the person interpreting them. LINT assumes the knowledge of an LLM, which may not reflect reality due to a mismatch of knowledge between the LLM and the target audience of the program. For example, if the goal of having interpretable programs is to teach people strategies for playing a real-time strategy game, then the LLM might have deeper knowledge of this genre of game than rookie players trying to learn strategies from the programs. As a result, a policy that is "interpretable" for the LLM is not necessarily interpretable to the target audience. Conversely, if the program requires knowledge that the LLM does not possess (e.g., π is written in a DSL different from the languages with which the LLM is trained), LINT can produce false negatives.

Similarly to the BLEU score [38], LINT should not be used as an objective function. Using LINT as such could cause the system to disregard C, and the explainer could generate non-interpretable explanations. Instead of using it as a target, LINT can be used as a tool to assess the interpretability of computer-written programs, to bias design decisions made during the development cycle of synthesizers.

Chapter 4

Empirical Methodology

The primary objective of our evaluation is to check whether the LINT scores correlate with the interpretability of a given set of programs. We rely on methods from the static obfuscation literature [12] to generate programs with different levels of interpretability. Static obfuscation algorithms have the goal of transforming a program before it starts running into less interpretable programs, with the goal of making it harder for adversarial agents to gain knowledge of the program by reading its implementation. For that, we consider semantics-preserving obfuscation transformations, where we can control the degree to which a program is obfuscated. We hypothesize that LINT scores correlate with the degree of obfuscation of a set of programs.

We consider two instances of LINT: one for evaluating the interpretability of programs that encode solutions to programming tasks; and another for evaluating programmatic policies [29] for playing MicroRTS, a real-time strategy game [35]. More information about the MicroRTS game can be found in the Appendix A.

The complete set of prompts used in our experiments can be found in the Appendix B (Sections B.2 and B.3). All experiments used GPT-4 [36]. We use $k = 5$ in all our experiments.

4.1 Classical Programming Problems

We consider 10 programs written in C for solving the following problems: computation of factorials, addition of two numbers, conversion of byte to bi-

```
1 void subsets(char *av[], int c, int n,
2 char *sbset[], int sz) {
3     if (c == n) {
4         if (sz < n) {
5             for (int i = 0; i < sz; i++)
6                 printf(sbset[i]);
7             printf("----------");
8         }
9         return;
10    }
11    sbsets(av, c+1, n, sbset, sz);
12    sbset[sz] = av[c];
13    sbsets(av, c+1, n, sbset, sz+1);}
14
15 main(Q,O)char**O;{if(--Q){main(Q,O);O[Q]
16 [O]^=0X80;for(O[O][O]=0;O[++O[O][O]]!=0;)
17 if(O[O[O][O]][O]>0)puts(O[O[O][O]]);
18 puts("----------");main(Q,O);}}
```

Figure 4.1: Non-obfuscated code for computing proper subsets (lines 1–13); an obfuscated program for the same problem (line 15).

nary, computation of all proper subsets of a set of arguments, of the value of π, of $\ln(n)$ for any n, of the smallest 100 prime numbers, of the square root of a number, sorting elements, and a program to play tic-tac-toe. The obfuscated versions of these programs were designed so that they would be as non-interpretable as possible, since all obfuscated programs we use are winning entries of the International Obfuscated C Code Contest [22]. The obfuscated programs were constructed using different techniques, such as replacing sequences of instructions with equivalents that are less interpretable [11]. Figure 4.1 shows an example of the programs used in our experiment, where the first function is a non-obfuscated implementation for computing the proper subsets of a set of numbers, while the second is an obfuscated implementation to solve the same problem. The proper subsets of a set I include all subsets except I. The complete set of programs is provided in the Appendix B (Sections B.6, B.4).

The function B we consider in this experiment measures the number of input values that the reconstructed program correctly maps to their corresponding output value. A B-value of 1.0 indicates that the reconstructed program mapped all inputs to the correct output; a value of 0.0 indicates that the reconstructed program failed on all inputs.

4.2 Programmatic Policies

We also used programmatic policies for playing MicroRTS. These programs are categorized into two types: "synthesized" policies, written by a computer program in the domain-specific language known as the Microlanguage [29], and "human-crafted" policies, written in Java by human programmers. Both types of programs receive a state of the game and return the action the agent performs in that state.

We consider the two-player version of MicroRTS, where each player controls a number of units to collect resources, build structures, and train other units that will eventually battle the opponent. Programmatic policies are the current state of the art in this domain, with programmatic policies winning the last three competitions.[1] MicroRTS has the following types of unit: Worker, Light, Ranged, Heavy, Base, and Barracks. The first four types can move around a gridded map where the game is played and attack opponent units; Workers can collect resources and build Bases and Barracks; Bases can train more Worker units and store resources, while Barracks can train non-Worker units. Units differ in how much damage they can inflict on opponent units and in how much damage they can suffer before being removed from the game. More details on MicroRTS game and its DSL can be found in Section A.1.

```
1 for(Unit u)
2     for(Unit u)
3         u.train(Worker,Up,2)
4     u.attack_if_in_range()
5     u.train(Heavy,EnemyDir,8)
6 for(Unit u)
7     u.train(Light,Left,100)
8     u.build(Barracks,EnemyDir,1)
9     u.harvest(25)
10    u.attack(Closest)
```

Figure 4.2: Policy written in the Microlanguage.

[1] https://sites.google.com/site/micrortsaicompetition

4.2.1 Microlanguage

The Microlanguage in a DSL for MicroRTS that allows programs to iterate through all units the player controls, so it assigns an action to each of the units. The language also supports if-then-else structures. Figure 4.2 shows an example of a program synthesized with Local Learner (2L) [33]. The loops allow for an action prioritization scheme. This is because once an action is assigned to a unit, it cannot be overwritten by another action, so the instructions in the earlier for-loops will be assigned first. In the program shown in Figure 4.2, training Worker units has the highest priority because the instruction for training these units is in the first nested for-loop to be executed (lines 2 and 3), which iterates through all units until it eventually finds a Base that will train them. Other actions that require the use of resources (e.g., constructing a Barracks in line 10), will be executed only if the player has enough resources after training Worker units.

4.2.2 Java

The human-crafted policies are written in Java and follow standard Java programming principles. This allows for the representation of more complex policies, but lacks the Microlanguage's specialized domain-specific approach. Figure 4.3 shows an example.

```
1 for (Unit u : pgs.getUnits())
2     if (u.getType() == barracks
3     && u.getPlayer()== player
4     && gs.getActionAssignment(u) == null)
5         if (p.getResources() >= light.cost)
6             train(u, light);
```

Figure 4.3: Policy written in Java by a human programmer.

4.2.3 Obfuscating Programmatic Policies

In the experiment with programmatic policies, we modified the programs to create different levels of interpretability, to verify whether the LINT scores

correlate with these levels. We achieve this using the obfuscation technique of adding useless snippets to the programs, which is a known program obfuscation technique [11]. We consider two levels of obfuscation: level 1, where we add a few lines of code that do not change the behavior of policy, and level 2, where we add a greater number of such lines compared to level 1. For the computer-synthesized set of policies, we add 10 and 23 lines for levels 1 and 2, respectively; for the human-crafted set, we add 38 and 71 lines for levels 1 and 2, respectively. Under the assumption that programs with longer useless snippets are less interpretable than programs with shorter snippets, we hypothesize that LINT assigns higher scores to non-obfuscated programs, lower scores to level 1, and the lowest scores to level 2.

Figure 4.4 shows a sample of a snippet that we add to the synthesized programmatic policies used in our experiments for level 1 of obfuscation; all snippets are shown in the Appendix B (Section B.5), including level 2 snippets. The snippet in Figure 4.4 does not change the behavior of the policy because the only unit that can harvest resources is a builder, so line 6 does not change the behavior of the policy. Additionally, if a unit is a worker, it is unable to train any other units (lines 12 and 14).

```
1 if (u.canHarvest()) then {
2     for (unit u){
3         if (u.isBuilder()) then{
4     }
5     else{
6         u.harvest(50);
7             }
8         }
9 }
10 for (Unit u){
11     if (u.is_Type(Worker)) then{
12         u.train(Heavy, Enemydir,10);
13         if (u.canAttack()) then {
14             u.train(Ranged,Up,16)
15         }
16     }
17 }
```

Figure 4.4: Sample of useless code snippet used in level 1.

4.2.4 Set of Policies Evaluated

For the synthesized set Π, we selected a subset of 20 programs from the totally ordered set with approximately $1,000$ programmatic best responses 2L synthesized for the BaseWorkers-16×16A map. Two adjacent policies in the ordered set are likely to be similar to each other due to the process in which 2L synthesizes them. We select 20 uniformly spaced policies from the ordered set to obtain a more diverse subset. That is, given that we have m policies in the ordered set, we select the policies with indices $\lfloor \frac{i \times (m-1)}{(19)} \rfloor$ with $i = 0, \cdots, 19$. For the human-crafted set, we used 10 programs selected from a collection available on GitHub[2]. We present all the programs used in our study in the Appendix B, Sections B.6 and B.4.

4.2.5 Behavior Metrics

We used three behavior metrics B for programmatic policies. For all metrics, we consider a set of 10 policies, which are chosen from a totally ordered set of programmatic policies 2L synthesized; we refer to this set as the set of opponents $\mathcal{O}$. Although the policies evaluated and the set of opponents are selected from the same pool of programs, there is no overlap between the two sets. We ensure that our metric results are not skewed by having overly weak or overly strong opponents. This is achieved by, while sampling policies for $\mathcal{O}$, rejecting those that win or lose all matches against the set of 20 policies we evaluate in our experiment. Let $S_{\pi,o}$ be the set of states in which the policy π is to act in a match played with the opponent o in $\mathcal{O}$. Also, let $S_\pi = \bigcup_{o \in \mathcal{O}} S_{\pi,o}$ be the union of the states of all matches played with the opponents.

4.2.5.1 Action Metric

The first metric, which can be applied to any sequential decision-making problem with discrete action spaces, is the fraction in which the actions chosen by the reconstructed program π' match the actions chosen by the original program π for states in the set S_π. This metric computes $|S_\pi|^{-1} \times \sum_{s \in S_\pi} \mathbf{1}[\pi(s) = \pi'(s)]$,

[2] `https://github.com/rubensolv/SCV/tree/master/pvai`

where $\mathbf{1}[\cdot]$ is the indicator function that equals 1 if the condition within the brackets is true, and 0 otherwise. We refer to this metric as the **action metric**. If the reconstructed program is equivalent to the original, then the action metric is 1.0.

Let us illustrate how to evaluate the action metric value between two programs, π and π', in the setting of a real-time strategy game through a comparative analysis. The process begins by running program π into a controlled game environment to face a predetermined opponent, labeled as o. We mark the game's initial state as s_0[3], at which point π selects an action, a_0. Following the selection of a_0, the game state for π transitions to s_1. However, we ensure to record state s_0 before moving to s_1. At this juncture, we introduce program π', allowing it to engage in the game from state s_0 and make its own strategic decision, resulting in action a'_0. This methodology enables a straightforward comparison of actions a_0 and a'_0 since both actions are based on the identical initial game state, s_0. The same evaluation process is replicated for subsequent states, such as s_1. For this next stage, s_1 is fixed and provided to both programs, π and π', enabling each to generate their respective actions, a_1 and a'_1.

This procedure is repeated, advancing through the game states until π and its opponent conclude the game. To obtain a comprehensive evaluation, this methodology is applied against a variety of opponents, with the results averaged to produce a mean action metric value.

To ensure a diverse comparison, an additional series of evaluations is initiated with program π' as the starting point. We observe the reactions of π within the states fixed by π', calculating the action metric from this perspective. The final measure of how similar the actions of the two programs are is calculated by taking the average of the action metric values from two sets of evaluation: those that use states generated from program π's gameplay and those from program π''s gameplay. You can see an illustrative image on this procedure (fixing π states) in Figure 4.5.

[3]Previously known as s_{init} in Section 2.5.2, now s_0 for more enumeration (e.g., s_1)

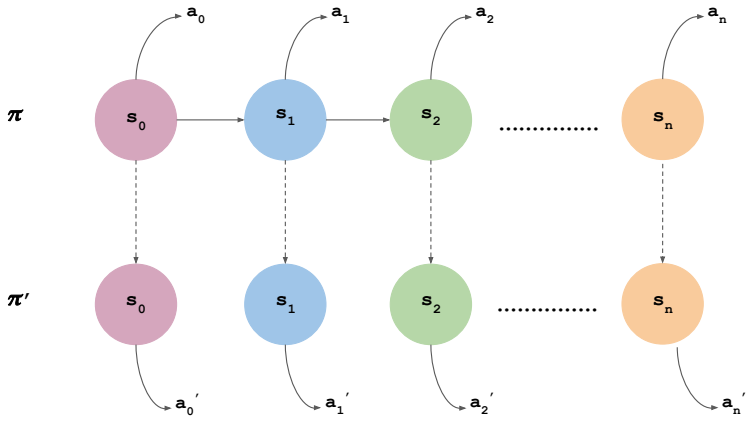

Figure 4.5: Computation of the Action Metric for Program Pair: This figure demonstrates the process for calculating the action metric between two strategies, π and π'. Actions denoted by a_x and a'_x represent the steps taken by each strategy at their respective states, starting from S_0 through S_n, with the states of program π held constant for comparison.

To accurately formulate the action metric value, considering evaluations starting from both programs π and π', we define two sets of states, S_π and $S_{\pi'}$, which represent the states encountered during the gameplay of π and π', respectively. The action metric value, denoted as $AMV_{avg}(\pi, \pi')$, is calculated as the average of the action metric values obtained from both sets of evaluations:

$$AMV_\pi = |S_\pi|^{-1} \sum_{s \in S_\pi} \mathbf{1}[\pi(s) = \pi'(s)] \tag{4.1}$$

$$AMV_{\pi'} = |S_{\pi'}|^{-1} \sum_{s \in S_{\pi'}} \mathbf{1}[\pi(s) = \pi'(s)] \tag{4.2}$$

$$AMV_{avg}(\pi, \pi') = \frac{AMV_\pi + AMV_{\pi'}}{2} \tag{4.3}$$

4.2.5.2 Outcome Metric

The second metric, which can be applied to any zero-sum game, compares the signature of wins, draws, and losses of the reconstructed policy with the signature of the original policy. The signature a_π of a policy π is a vector of size $|\mathcal{O}|$ where each entry i assumes the values of 1, 0, or -1, representing the

result of a win, draw, or loss, respectively, of a match played between π and the i-th opponent in $\mathcal{O}$. This metric computes $|\mathcal{O}|^{-1} \times \sum_{i=1}^{|\mathcal{O}|} \mathbf{1}[a_\pi[i] = a_{\pi'}[i]]$, where $a_\pi[i]$ represents the i-th entry of a_π. We call this metric the **outcome metric**. Similarly to the action metric, if π' is equivalent to π, then the outcome metric value is 1.0.

4.2.5.3 Feature Metric

The third metric compares the set of features observed in matches between the reconstructed program and $\mathcal{O}$ with the features observed in matches between the original program and $\mathcal{O}$. Let $F(\pi, o)$ be a vector of features observed in a match between π and o. We use the seven features of Alexio et al. [2], where each feature is the sum of the number of units of a given type that the player trained (or built) in all states of the match; the types can be Worker, Light, Heavy, Ranged, Base, or Barracks. A last feature sums up the amount of resources collected in the match. This metric measures the average normalized L1 norm between the feature vector of the original and reconstructed programs. We refer to this metric as **feature metric**. If the reconstructed program is equivalent to the original, then this metric is 0.0. While other metrics measure similarity, the feature metric measures dissimilarity.

We use these three metrics because each of them individually has weaknesses; together, they offer a more reliable summary of the behavior of a policy. Many of the actions in a MicroRTS match are related to Worker units collecting resources, so while two policies might encode totally different strategies, due to the large number of Worker units collecting resources, the policies could have a action metric value close to 1.0. The outcome metric can also be misleading if almost any policy defeats the set of opponents or is defeated by the set of opponents (i.e., the opponents are too weak and/or too strong). Finally, the feature metric simply counts the number of units and resources, without measuring their behavior.

4.2.6 Baselines for Reconstructed Programs

We consider a number of programs as baselines for the programs LINT reconstructs. Namely, for each program π in Π, we compare the behavior metric values for the reconstructed program π' of π with a randomly selected program from Π that is different from π; we call this baseline **Rand**. In the experiment with the synthesized set, since all programs in Π were generated in a single run of 2L and for a fixed map, the programs Rand selects can be similar to π.

In another baseline, where we select a random program from the pool of programs 2L synthesizes for a different map; we use programs synthesized for the BaseWorkers-8×8 map with this baseline. Since the strategy for playing the game can change drastically from map to map, this randomly selected program is likely to be less similar to π than the programs Rand selects. We call this baseline **Rand-Other**.

Another baseline we consider selects the policy from the set of evaluated policies Π that is different from π but is most similar to π with respect to its syntax. We treat each line of a program as an element of a set. The program most similar to π is the one whose intersection with the syntax set of π is the largest. We refer to this baseline as **Closest-Syntax**.

The last baseline we consider returns the policy in Π different from the evaluated program π with the smallest L2 norm for the difference between its feature vector and the feature vector of π. We refer to this baseline as **Closest-Feature**.

4.2.7 Baseline for the Reconstruction System

The Reconstructor relies on the natural language explanation of the original program that the Explainer provides. To ensure that the program cannot be reconstructed by sampling programs without the explanation, we consider a baseline where we ask an LLM to generate a program for playing on a given map of MicroRTS. We provide a natural language description of the BaseWorkers-16×16A map and ask the model to generate a program written in the Microlanguage that can "play well" the game. We repeat this process k

times, exactly how it is implemented in the reconstruction system for LINT, and we return the program with the best behavior score out of the k evaluated. We refer to this baseline as the **k-Shot LLM**. This approach acts as an ablation study for the explanation component.

We hypothesize that behavior scores for this baseline will be worse than the ones obtained for the reconstructed program, since the reconstruction is more accurate if conditioned on the natural language explanation the Explainer provides.

Chapter 5

Empirical Results

In this chapter, we explore the empirical results from two domain: **Classical Programming Problems written in C language**, and **Programmatic Policies**. The discussion on Programmatic Policies is further branched into two subsets: those generated by computer, known as the **synthesized set**, and those manually created by individuals, referred to as the **human-crafted set**.

5.1 Classical Programming Problems

The system was able to reconstruct all non-obfuscated programs to solve the programming problems and achieved the score of 1.0. The system was unable to reconstruct any of the obfuscated programs, resulting in a score of 0.0. As a representative example, Explanation 1 shows the Explainer's output for the non-obfuscated program shown in Figure 2.1.

Explanation 1 (Proper Subset - Non-Obfuscated) *Imagine you have a list of items, and you want to figure out all the different ways you can select some of these items without taking all of them at once. When the program starts, it first checks if you've given it any items to work with. If not, it asks for some. Once it has the list of items, it begins to explore all the possible selections. It does this by starting with the first item and deciding whether to include it in the current selection or not. If it decides not to include the first item, it moves to the next item and makes the same decision. It continues*

doing this until it has considered every item in the list. After considering all the items, if it has left out at least one item from the current selection, it displays that selection for you to see. Then, it goes back and tries a different combination by including some items that it previously left out and leaving out some that it previously included. It continues this process until it has explored and shown you all possible selections of items from the list, always making sure it never shows you a selection that includes all the items at once. In the end, you're left with a display of all the different ways you can select some of the items without taking all of them.

The explanation of the program is high-level and makes an analogy with a list of items, and the process of computing the proper subsets is described as a process of "selecting" and "displaying" the items, where all possible combinations of subsets are selected: *"It does this by starting with the first item and deciding whether to include it in the current selection or not"*. The Reconstructor can produce a correct implementation of a program that computes the proper subsets from this explanation. Explanation 2 shows the explanation LINT generated for the obfuscated program in Figure 2.2.

Explanation 2 (Proper Subset - Obfuscated) *Imagine you have a list of items. Initially, all items are marked as "normal." The program goes through the following steps:*

1. *It starts at the end of the list and works its way to the beginning. For each item, the program does two actions:First, it keeps the item as "normal" and then displays all the items marked as "greater than normal." After displaying these items, it separates them with a line of dashes. Second, it switches the item from "normal" to "greater than normal" and repeats the display process.*

2. *After dealing with an item, the program moves to the next item closer to the beginning of the list and repeats step 2.*

This process continues until the program has considered all items in the list.

This explanation is well-structured and includes the steps that supposedly need to be performed. However, the description is not clear in some parts. For example, it is not clear what "greater than normal" means.

5.2 Programmatic Policies

Table 5.1 shows the average and 95% confidence interval values for the 20 programs used in our experiment with the synthesized set. The LINT row shows the metric values computed for the original programs and their reconstructions. The baseline values represent the measurements between the original programs and the baseline programs. The values of LINT are the best according to all metrics, which shows that the reconstructed programs are more similar to the original than any of the baselines. Table 5.1 also shows that the baselines that obtain values closer to LINT are Rand, Closest-Syntax, and Closest-Feature. This is because these baselines select a program from the pool of programs 2L synthesized for the same map, and these programs tend to be similar to each other. Rand-Other obtained lower Action and Outcome values and a higher Feature value, demonstrating that the metrics can capture the expected differences between policies synthesized for playing in different maps. Finally, k-Shot LLM presents the lowest similarity values, demonstrating the importance of the explanation of the original program the Explainer generates.

	Metrics B		
	Action (↑)	Outcome (↑)	Feature (↓)
LINT	0.945 ± 0.010	0.840 ± 0.042	0.133 ± 0.020
Rand	0.733 ± 0.015	0.615 ± 0.057	0.418 ± 0.018
Rand-Other	0.564 ± 0.025	0.470 ± 0.058	0.486 ± 0.012
Closest-Syntax	0.799 ± 0.015	0.600 ± 0.056	0.403 ± 0.018
Closest-Feature	0.823 ± 0.013	0.770 ± 0.049	0.189 ± 0.014
k-Shot LLM	0.343 ± 0.010	0.420 ± 0.057	0.441 ± 0.009

Table 5.1: Average value of the behavior metrics for synthesized programmatic policies for LINT and other baselines. Action and Outcome metrics are similarity metrics, so higher values are better (↑), while Feature is a metric of dissimilarity, so lower values are better (↓). The cells show the metric values and the 95% confidence interval.

Table 5.2 presents the results that test our hypothesis that LINT correlates with the degree of interpretability of the programs. The results indicate a higher similarity between the original and reconstructed programs for the policies 2L synthesizes than between the original obfuscated programs and their reconstructions. Also, LINT provides higher similarity scores and a lower dissimilarity score for Level 1 than for Level 2 obfuscation.

Metric	Original Program	Obfuscation Level	
		Level 1	Level 2
Action	0.945 ±0.010	0.866 ±0.014	0.732 ±0.012
Outcome	0.840 ±0.042	0.705 ±0.053	0.490 ±0.058
Feature	0.133 ±0.020	0.272 ±0.024	0.418 ±0.018

Table 5.2: Average LINT values of the behavior metrics for the original synthesized program and for the two levels of obfuscation; the cells also show the 95% confidence interval. In this table, darker cells indicate that the reconstructed program is more similar to the original one.

Table 5.3 presents the results for the human-crafted set. These results align with those of the synthesized set, where the LINT-score decreases as we increase the level of obfuscation. Under the assumption that higher degrees of obfuscation result in less interpretable programs, the results support our hypothesis.

Metric	Original Program	Obfuscation Level	
		Level 1	Level 2
Action	0.98 ±0.003	0.92 ±0.017	0.85 ±0.031
Outcome	0.95 ±0.036	0.84 ±0.060	0.80 ±0.066
Feature	0.07 ±0.022	0.19 ±0.029	0.22 ±0.031

Table 5.3: Average LINT values of the behavior metrics for the original program in the human-crafted set and for the two levels of obfuscation; the cells also show the 95% confidence interval. In this table, darker cells indicate that the reconstructed program is more similar to the original one.

Figure 5.1 offers a visual interpretation of the data presented in Table 5.1, showcasing the same data points in a distinct format. While they show a similar trend, they're not precisely correlated, highlighting the individual

weaknesses of the metrics. This variation emphasizes the importance of using these metrics together for a thorough analysis of interpretability.

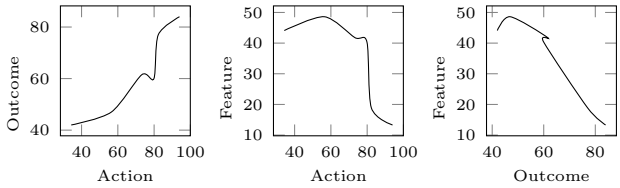

Figure 5.1: Comparative analysis of different metrics in relation to Action, Outcome, and Feature. The plots showcase relationships between (a) Action and Outcome, (b) Action and Feature, and (c) Outcome and Feature from Table 5.1.

5.3 Representative Sample

In this section, we present the natural language explanation the Explainer generated for the program in Figure 5.2 (Explanation 3). Then, we present the program the Reconstructor generates from the explanation (Figure 5.3). This is a representative example of the programs used in the experiments.

```
1 for(Unit u)
2     for(Unit u)
3         u.train(Worker,Up,2)
4     u.attack_if_in_range()
5     u.train(Heavy,EnemyDir,8)
6 for(Unit u)
7     u.train(Light,Left,100)
8     u.build(Barracks,EnemyDir,1)
9     u.harvest(25)
10    u.attack(Closest)
```

Figure 5.2: Policy written in the Microlanguage.

Explanation 3 (Programmatic Policy) *The script outlines a two-tiered strategy in the game of MicroRTS. It focuses on the tactical management of different types of units, ranging from Workers, to Heavy units, and finally Light units. The first part of the strategy involves a layered assignment of tasks. Every unit is asked to train two Worker units if possible. These Workers can be seen as the backbone of your army since they are the only units that*

can build structures, which in turn produce other units. At the same time, if a unit can't train a Worker, it is assigned to remain idle, thus serving as a defensive guard to your base. In addition, every unit is asked to train up to 8 Heavy units, that typically are stronger and can inflict more damage, acting as the frontline offensive unit. This phase essentially sets up your base with a balance of productive Workers and Heavy units.

In the second part of the strategy, each unit is tasked with producing as many Light units as possible, up to a whopping 100. Light units are generally quicker and can swiftly react to the changes in the battlefield. Additionally, every unit is tasked to build a Barracks near the enemy's direction, effectively establishing a forward base closer to the enemy territory. They are also asked to harvest resources, up to 25 units, ensuring a steady supply for creating more units and structures. Finally, units are directed to attack the closest enemy units, marking an aggressive stance towards the opponent.

The overall goal of this strategy is to ensure a robust base with a mixture of Worker and Heavy units, while also maintaining an aggressive stance with a large army of Light units and Barracks near the enemy territory. It aims at resource gathering for continued production of units and structures, and pushing the opponent back through relentless attack.

Similarly to Explanation 1, the explanation describes the policy in a level of detail that allows for the reconstruction of a program that behaves identically to the original program according to our metrics. Figure 5.3 shows the reconstructed program. The reconstructed program is not identical in terms of line-by-line syntax, since information regarding the syntax is lost in the process of translating the program into a natural language explanation and back to a program. A casual inspection might even suggest that the reconstructed program does not behave as the original. This is because the original program has an instruction for training Worker units inside a nested loop, thus giving it the highest priority. The reconstructed program has training Worker units instruction inside the main loop. This means that the program can use the player's resources to assign actions to other units (e.g., train Light units in

```
1 for(Unit u):
2     u.train(Worker, Up, 2)
3     u.attack_if_in_range()
4     for(Unit u):
5         u.train(Heavy, EnemyDir, 8)
6     for(Unit u):
7         u.train(Light, Left, 100)
8     u.build(Barracks, EnemyDir, 1)
9     u.harvest(25)
10    u.attack(Closest)
```

Figure 5.3: Reconstruction of the program shown in Figure 4.2.

line 7) and by the time u is a Base in the outer loop, the player no longer has resources for training Worker units. However, a more careful inspection of the program reveals that, in the first states of the game, where the player trains Worker units, none of the actions that use resources can be assigned to a unit: the player cannot train Heavy and Light units (lines 5 and 7, respectively) because the player does not have a Barracks yet; the player cannot build a Barracks (line 8) because it does not have enough resources to do so. Thus, similarly to the original program, the reconstructed one prioritizes the training of Worker units.

Chapter 6

Discussion, Future Works, and Takeaways

In this chapter, we will explore the limitations of LLMs, identify the constraints of our current study, and consider potential avenues for future research related to this work.

6.1 Caveats of LLMs

6.1.1 Data Contamination

Data contamination in LLMs refers to the phenomenon in which models are inadvertently trained or influenced by the data they were tested on, possibly compromising the integrity of the evaluation. Sources of contamination include direct training on benchmark datasets and indirect exposure via user interactions, especially in closed-source models like ChatGPT [5].

6.1.1.1 Direct data Leakage

GPT-4's training data includes information up until September 2021. This means it encompasses a wide range of knowledge up to that point, but it does not include information on events or developments occurring after September 2021.

Based on this information, it's crucial to verify whether the data used in our evaluations was part of GPT-4's training material. We need to check if explanations of policies and the reconstructed program were accessible online

before GPT-4's training cutoff. To the best of our knowledge, no such information was available online before GPT-4 was trained. Additionally, our paper was published on arXiv in October 2023 for the first time. Therefore, if a new version of GPT is trained up to that date, we'll need to be cautious about the possibility of our results being compromised by direct data leakage. However, for now, this concern does not apply.

6.1.1.2 Indirect Data Leakage

Indirect data leakage in LLMs, particularly highlighted in web interfaces like ChatGPT occurs when models are continually updated with data stemming from user interactions through RLHF (Reinforcement Learning from Human Feedback) [37]. This process integrates new data into the model and the model is fine-tuned based on this data just by virtue of its use, constituting a form of data leakage not directly involving the training set. Therefore, believing that "using benchmarks only accessible to authorized individuals, or datasets developed after the release of ChatGPT, ensures they have not been leaked" is a mistaken notion.

We refer to OpenAI's data usage policy to confirm this. As you can see, they explicitly mentioned the use of users' data for model training:

OpenAI's Data Usage Policy

> "[...] when you use our services for individuals such as ChatGPT or DALL-E, we may use your content to train our models [...]"

They also explained that when users send data through API and business services, this information isn't used to train the model:

OpenAI's Data Usage Policy

> "[...] we don't use content from our business offerings [...] and our API Platform to train our models [...]"

Therefore, only interactions with the models through the web interface are regarded as potential data leaks [5].

Returning to our work, we conducted initial experiments using the web interface (ChatGPT) to validate our approach before scaling up with a larger number of experiments via API. This means we may have been exposed to potential indirect data leakage, although the extent of this exposure is uncertain.

Now, let us delve into where this leakage could occur within our work. We have three components: the explainer, reconstructor, and verifier. In the explainer, we input the DSL, program, and instructions to generate an explanation. The reconstructor takes the DSL, explanation, and instructions to regenerate the code. Notably, there is no direct link between the data and the label we seek. However, in the verifier, we provide both the explanation and the reconstructed program in a prompt, which could potentially expose the data-label relationship to the LLM during a query.

Despite this, we found our initial prompts to be promising, and we did not observe improved performance over time, which gives us some confidence that leakage may not have significantly influenced our results.

For future work, we could explore downloading the weights of another LLM, such as LLaMA [48], and running our system locally. We hypothesize that we would achieve similar results, providing further validation of our approach.

6.1.2 Hallucination

Hallucination in the context of LLMs refers to instances where these models generate information or data that is not grounded in the input provided or factual reality. This means they might produce statements that are misleading, incorrect, or completely made up. Hallucination poses significant challenges, especially in applications requiring high accuracy and reliability, such as in the fields of medicine, law, and news reporting.

Several techniques can be employed to minimize the occurrence of hallucination in LLMs. These include techniques such as Prompt Engineering, Retrieval Augmented Generation (RAG) [28], and Feedback and Reasoning Mechanisms among others. In our approach, we focused on refining prompt engineering techniques and conducted multiple iterations of our experiments. By repeating the experiments several (k) times, we aimed to ensure that our

findings were robust and not the result of random chance or hallucinations by the model.

6.2 Limitations of Our Study Concerning Programming Problems

Our outcome metric for programming problems (the ones written in C) quantifies the proportion of passed test cases, with results ranging from 0 to 1.

In our set of obfuscated and their original C problem sets, we only have examples that are either highly obfuscated or not obfuscated at all, resulting in LINT scores of 0 and 1, respectively. We need to include lightly-obfuscated programs that can generate error-prone explanations. These programs would yield near-correct outputs and likely pass some, but not all, test cases.

We also encounter scenarios where a program might seem interpretable, like one that sorts numbers. A person reviewing the program might find it interpretable and believe it deserves a high score for interpretability. However, if a bug prevents it from passing any test cases, our system assigns it a 0, falsely indicating poor interpretability.

Our study does not yet include such nuanced examples, but they should be considered in future research to potentially develop a more nuanced similarity metric specifically for non-MDP problems that rely solely on the outcome metric.

6.3 Future Directions for Research

One promising avenue for extending this research involves conducting a user study. While our research is grounded in both the literature on code obfuscation and analysis of human-written code, affirming the validity of our claims and the empirical support for our findings, we acknowledge the importance of further validation through a user study. Our hypothesis suggests that there will be a correlation between the LINT score and the outcomes derived from user-study evaluations, indicating the reliability of the LINT score as an interpretability measure.

Additionally, future research could explore using different LLMs aside from GPT-4, by experimenting with various configurations, such as distinct LLM types for explainer and reconstructor components. Another avenue for investigation is the optimization of prompt design, aiming to identify the most concise yet effective version of prompts that could maintain or enhance performance. Finally, the issue highlighted in Section 6.2 deserves further exploration in upcoming research.

Chapter 7

Conclusion

Programmatic policies are often synthesized with the expectation of interpretability. However, to our knowledge, there has not been a systematic evaluation of the interpretability of such policies, probably due to the cost associated with such an evaluation. Especially because the evaluation of programmatic policies might require human users proficient in computer programming. In this dissertation, we presented an inexpensive methodology based on LLMs to assess the interpretability of programmatic policies. Namely, we introduced the LLM-based Interpretability (LINT) score for programs. The LINT score of a program is computed by having an LLM generate a description of it in natural language, which is provided as input to another LLM. This second LLM tries to reconstruct the program from the natural language description. The LINT score measures the similarity between the original and reconstructed programs. Our empirical evaluation of LINT relied on the literature on program obfuscation and we assumed that obfuscated programs are less interpretable than non-obfuscated ones. Empirical results on programming problems and programmatic policies showed that the LINT scores of the evaluated programs correlate with their interpretability. Our results suggest that LINT can be used as a tool to assess the interpretability of programmatic policies.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, Springer, 2013, pp. 934–950.

[2] D. S. Aleixo and L. H. S. Lelis, "Show me the way! Bilevel search for synthesizing programmatic strategies," in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 2023.

[3] R. Alur, R. Bodik, G. Juniwal, *et al.*, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[4] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*, Springer, 2017, pp. 319–336.

[5] S. Balloccu, P. Schmidtová, M. Lango, and O. Dušek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," *arXiv preprint arXiv:2402.03927*, 2024.

[6] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Proceedings of the International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2018, pp. 2499–2509.

[7] U. Berger, "Brown's original fictitious play," *Journal of Economic Theory*, vol. 135, no. 1, pp. 572–578, 2007, ISSN: 0022-0531. DOI: https://doi.org/10.1016/j.jet.2005.12.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022053106000299.

[8] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[9] T. B. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[10] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, pp. 546–558, 2010.

[11] F. B. Cohen, "Operating system protection through program evolution," *Computer Security*, vol. 12, no. 6, pp. 565–584, 1993.

[12] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[13] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, Association for Computing Machinery, 2015, pp. 107–118, ISBN: 9781450336758.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] Farama Foundation, *Microrts game definition*, 2023. [Online]. Available: `https://github.com/Farama-Foundation/MicroRTS/wiki/Game-Definition`.

[16] L. Ferreira, L. Lelis, and J. Whitehead, "Computer-generated music for tabletop role-playing games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, 2020, pp. 59–65.

[17] J. R. Glenn and C. J. Aloi, "A generalized heuristic for can't stop," in *The Florida AI Research Society*, 2009. [Online]. Available: `https://api.semanticscholar.org/CorpusID:9192452`.

[18] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

[19] I. Husien and S. Schewe, "Program generation using simulated annealing and model checking," in *International Conference on Software Engineering and Formal Methods*, Springer, 2016, pp. 155–171.

[20] J. P. Inala, O. Bastani, Z. Tavares, and A. Solar-Lezama, "Synthesizing programmatic policies that inductively generalize," in *International Conference on Learning Representations*, 2020. [Online]. Available: `https://openreview.net/forum?id=S1l8oANFDH`.

[21] *Introduction to microrts ai competition*, `https://sites.google.com/site/micrortsaicompetition/introduction?authuser=0`, 2023.

[22] IOCCC. "International obfuscated c code contest." Accessed: 2023-08-11. (1984), [Online]. Available: `https://www.ioccc.org/`.

[23] M.-Y. Kim, S. Atakishiyev, H. K. B. Babiker, *et al.*, "A multi-component framework for the analysis and design of explainable artificial intelligence," *Machine Learning and Knowledge Extraction*, vol. 3, no. 4, pp. 900–921, 2021, ISSN: 2504-4990. DOI: 10.3390/make3040045. [Online]. Available: https://www.mdpi.com/2504-4990/3/4/45.

[24] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and computing*, vol. 4, pp. 87–112, 1994.

[25] M. Lanctot, V. Zambaldi, A. Gruslys, *et al.*, "A unified game-theoretic approach to multiagent reinforcement learning," *Advances in neural information processing systems*, vol. 30, 2017.

[26] T. László and A. Kiss, "Obfuscating c++ programs via control flow flattening," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:5061467.

[27] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," *ACM SIGPLAN Notices*, vol. 53, pp. 436–449, Jun. 2018. DOI: 10.1145/3296979.3192410.

[28] P. Lewis, E. Perez, A. Piktus, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[29] J. R. H. Mariño, R. O. Moraes, C. F. M. Toledo, and L. H. S. Lelis, "Evolving action abstractionsfor real-time planning in extensive-form games," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[30] L. C. Medeiros, D. S. Aleixo, and L. H. S. Lelis, "What can we learn even from the weakest? Learning sketches for programmatic strategies," in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 2022, pp. 7761–7769.

[31] L. C. Medeiros, D. S. Aleixo, and L. H. S. Lelis, *What can we learn even from the weakest? learning sketches for programmatic strategies*, 2022. arXiv: 2203.11912 [cs.AI].

[32] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial intelligence*, vol. 267, pp. 1–38, 2019.

[33] R. O. Moraes, D. S. Aleixo, L. N. Ferreira, and L. H. S. Lelis, *Choosing well your opponents: How to guide the synthesis of programmatic strategies*, 2023. arXiv: 2307.04893 [cs.LG].

[34] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 348–359. DOI: 10.1109/ICSME46990.2020.00041.

[35] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes, and L. H. S. Lelis, "The first microrts artificial intelligence competition.," *AI Magazine*, vol. 39, no. 1, 2018.

[36] OpenAI, : J. Achiam, *et al.*, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[37] L. Ouyang, J. Wu, X. Jiang, *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[38] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 2002, pp. 311–318.

[39] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the Working Conference on Mining Software Repositories*, Association for Computing Machinery, 2011, pp. 73–82, ISBN: 9781450305747.

[40] W. Qiu and H. Zhu, "Programmatic reinforcement learning without oracles," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=6Tk2noBdvxt.

[41] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245.

[42] A. Ramesh, M. Pavlov, G. Goh, *et al.*, "Zero-shot text-to-image generation," in *International conference on machine learning*, Pmlr, 2021, pp. 8821–8831.

[43] A. Razavi, A. Van den Oord, and O. Vinyals, "Generating diverse high-fidelity images with vq-vae-2," *Advances in neural information processing systems*, vol. 32, 2019.

[44] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 595–613, 2021. DOI: 10.1109/TSE.2019.2901468.

[45] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *IEEE International Conference on Program Comprehension*, 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503707.

[46] A. Solar-Lezama, "The sketching approach to program synthesis," in *Asian symposium on programming languages and systems*, Springer, 2009, pp. 4–13.

[47] A. Solar-Lezama, *Lecture 1: Introduction to program synthesis*, https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture1.htm, Accessed: 2024-02-28, 2023.

[48] H. Touvron, T. Lavril, G. Izacard, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[49] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 287–296, ISBN: 9781450320146. DOI: 10.1145/2491956.2462174. [Online]. Available: https://doi.org/10.1145/2491956.2462174.

[50] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[51] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[52] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 5045–5054. [Online]. Available: https://proceedings.mlr.press/v80/verma18a.html.

[53] J. Wei, X. Wang, D. Schuurmans, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

Appendix A

A.1 An Overview of MicroRTS

MicroRTS is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

A.1.1 Units

Units in MicroRTS can be categorized into several types, each with specific roles and capabilities.

Worker Units

Worker units have the ability to:

- Collect resources
- Build structures (Barracks and Bases)
- Attack opponent units

Barracks and Bases

- Barracks can train combat units (they can produce Light, Heavy, or Ranged units).
- Bases can train the Workers (they can only produce Workers).
- They can neither attack opponents' units nor move.

Combat Units

Combat units can be of type Light, Heavy, or Ranged. These units differ in:

- How long they survive a battle
- How much damage they can inflict to opponent units
- How close they need to be from opponent units to attack them

Resource Units

Resource units are the source of resources and do not belong to any player. These units cannot execute any actions. When the number of resources left reaches 0, the unit disappears. It only has one parameter: resources left.

A.1.2 Gameplay

Actions in MicroRTS are deterministic and there is no hidden information. A match is played on a map and each map might require a different strategy for defeating the opponent.

Fig /refmicrorts shows an example of a MicroRTS game state: gray circle units represent the Worker units, cyan circle units are the Ranged units and yellow circle units represent Heavy units. Light green squares represent the resources to be collected by the Workers, white squares are the Bases and gray squares are the Barracks.
A sample strategy for playing this game, specifically tailored for the 16x16 map, is presented below.

```
def Sketch-SA-O-16x16(state s):
    for u in s:
        u.train(Ranged, Right, 4)
        u.harvest(9)
        u.attack(Closest)
        u.train(Worker, Up, 1)
```

Figure A.2: A sample strategy for playing MicroRTS for a 16x16 map.

[0]Source: https://sites.google.com/site/micrortsaicompetition/microrts

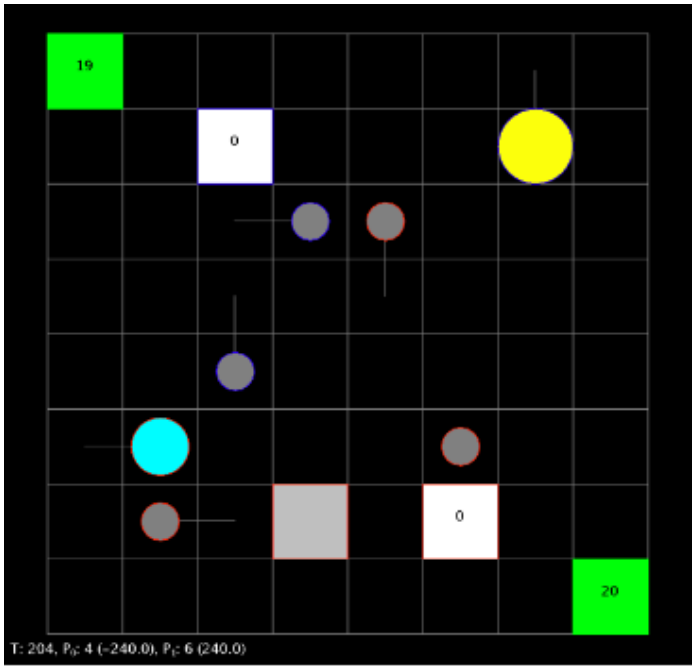

Figure A.1: A MicroRTS match. Different outline colors represent different players.

The strategy accepts a state s and allocates an action to each unit u contained in s. If a unit u is not assigned an action, it will not perform any action in the following round of the game. Once an action is allocated to a unit u, this decision is irreversible and the action cannot be changed.

Actions are designated based on the unit type. For instance, for a Base unit, the only applicable action is $u.train(Worker, Up, 1)$, indicating that Base units are exclusively responsible for training workers.

Taking another example, for a worker unit, the applicable commands are limited to $u.harvest(9)$ and $u.attack(Closest)$. It is important to note that operations proceed in a sequential order, and a unit cannot be assigned more than one action. Specifically, the $u.harvest(9)$, command deploys 9 worker units to gather resources. Once these 9 workers are allocated to harvesting, any remaining workers are then assigned the $u.attack(Closest)$ command, directing them to engage the nearest enemy units. [2], [15], [21], [35]

A.1.3 MicroRTS' DSL

The following describes a scripting language for playing the real-time strategy game called Micro-RTS.

Command-Oriented Functions

Here is a list of command-oriented functions used in the Micro-RTS DSL: (these instructions assign actions to units!)

- **Build(T, D, N):** Builds N units of type T on a cell located in the D direction of the unit.
- **Train(T, D, N):** Trains N units of type T on a cell located in the D direction of the structure responsible for training them.
- **moveToUnit(T_p, O_p):** Commands a unit to move towards the player T_p following a criterion O_p.
- **Attack(O_p):** Commands a unit to attack units of the opponent player following a criterion O_p.
- **Harvest(N):** Sends N Worker units to harvest resources.
- **attackIfInRange():** Commands a unit to stay idle and attack if an opponent unit comes within its attack range.
- **MoveAway():** Commands a unit to move in the opposite direction of the player's base.

Boolean Expressions

And here is a list of Boolean expressions each evaluating something at a given state:

- **HasNumberOfUnits(T, N):** Checks if the ally player has N units of type T.
- **OpponentHasNumberOfUnits(T, N):** Checks if the opponent player has N units of type T.

- **HasLessNumberOfUnits(T, N):** Checks if the ally player has less than N units of type T.
- **HaveQtdUnitsAttacking(N):** Checks if the ally player has N units attacking the opponent.
- **HasUnitWithinDistanceFromOpponent(N):** Checks if the ally player has a unit within a distance N from an opponent's unit.
- **HasNumberOfWorkersHarvesting(N):** Checks if the ally player has N units of type Worker harvesting resources.
- **is_Type(T):** Checks if a unit is an instance of Type T.
- **IsBuilder():** Check if a unit is of type Worker.
- **CanAttack():** Checks if a unit can attack.
- **HasUnitThatKillsInOneAttack():** Checks if the ally player has a unit that kills an opponent's unit with one attack action.
- **OpponentHasUnitThatKillsUnitInOneAttack():** Checks if the opponent player has a unit that kills an ally's unit with one attack action.
- **HasUnitInOpponentRange():** Checks if a unit of the ally player is within attack range of an opponent's unit.
- **OpponentHasUnitInPlayerRange():** Checks if a unit of the opponent player is within attack range of an ally's unit.
- **CanHarvest():** Checks if a unit can harvest resources.

Appendix B

B.1 Overview

The prompts for explanation, reconstruction, and verification of both *C Programming Problems* and *Programmatic Policies* (either Synthesized or Human-crafted) are detailed in the subsequent sections. The C Programming Problems and Human-crafted set of policies are developed in C and Java, respectively, whereas the Synthesized set is written in MicroLanguage

The distinction between the two groups is based on the language in which they're implemented. For *C Programming Problems* and *Human-crafted Java Policies*, there's no need to clarify any domain-specific language since the programs are written in a language that the LLM is already acquainted with its terminology. On the other hand, the *Synthesized Policies* utilize a language unique to MicroRTS. As a result, it's necessary to define this language and its components to the LLM. Specifically, for the explanation prompt, we must elucidate the DSL to ensure the LLM comprehends the program it's about to explain. Similarly, the DSL must be defined during reconstruction to ensure the regenerated program follows the MicroRTS programming rules. The same applies to the verifier.

B.2 MicroRTS Prompts

B.2.1 Synthesized Set

B.2.1.1 Explainer Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:
     - how long they survive a battle
     - how much damage they can inflict to opponent units
     - how close they need to be from opponent units to attack them.
  3. They can attack the oppponent units.
- Resource units:
  1. The source of resources, doesn't belong to any player.
  2. These units cannot execute any actions.
  3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.

This CFG allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units. The following describes a scripting language for playing MicroRTS.
The Boolean functions are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.
2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.
3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.
4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.
5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.
6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.
7. u.is_Type(T): Checks if a unit is an instance of type T.
8. u.isBuilder(): Checks if a unit is of type Worker.
9. u.canAttack(): Checks if a unit can attack.
10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.
11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action. v u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.
12. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.
13. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions are described below. These functions assign actions to units.

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.idle(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base. 'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions. 'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players. 'e' is an empty block which means doing nothing.

The for loops in this scripting language iterate over all units and the instructions inside the for loops attempt to assign actions to each of these units. For example, for (Unit u) u.build(Barracks, EnemyDir, 8)
The snippet above will assign a build action to unit u. Note that the only unit that can build is Worker. In the for loop above, if u is Ranged, for example, then the instruction u.build(Barracks,EnemyDir,8) will be ignored. Once an action is assigned to a unit, it cannot be changed. That is why the for loops offer a priority scheme to the actions. The way that for loop are organized is perhaps the most important feature of this language. The program always has the form: for (Unit u) ... With the possibility of adding nested for-loops that go through all units. The parameter in each instruction limits the amount of the thing that is trained or build. For example, u.build(Barracks, EnemyDir, 8) limits to at most 8 Barracks. If the player already has 8 barracks, then this instruction will be ignored. Now, you have the background you need to know!
Now, given the above information, first try to understand the meanings of all the boolean (B) and command (C) functions from above and try to relate them in the context of microRTS playing strategies.
Alright, let's take a look at program P in which I want you to write an explanation for:

Program P

The following 7 are some guidelines for writing an explanation for this strategy:

1. Please write a high-level explanation and do not explain the code line by line but try to include numbers and unit names in your natural language explanation.
2. Try to understand what is happening in the code and explain it in natural language for someone who wants to know how to play MicroRTS using this strategy.
3. Write the explanation inside '$< explanation >< /explanation >$' tag.
4. DON'T USE any quotation marks in writing the explanation.
5. At the end of the explanation, write the overall goal of the strategy as well. (include it inside the explanation tag)
6. Don't forget to mention the numbers but in a natural language way.
7. The for-loops offer a hierarchy that determines the priority of the actions.

The list below specifies the different priorities for actions one can obtain with the for-loops (from highest to lowest priority).

- Actions inserted in nested for-loops at the top of the program receives the highest priority.
- Actions inserted in nested for-loops that appear later in the program have higher priority than actions that appear outside a nested for loop.
- Actions outside the nested for-loops that appear earlier in the program have higher priority than actions outside for-loops that appear later in the program.

The most important part of the explanation is to be clear with respect to the priority of the actions. That is, what is the action with highest priority, which one follows that one, and so on. You MUST NOT talk in terms of for-loops and programming language jargon as people not familiar with programming will not understand the explanation. Also, you MUST not use any 'nest' or 'nested' words as they are related to programming elements. You should talk in terms of priorities of the actions, as you would explain the strategy to a gamer.
For example, if an action is within a nested for loop, then you need to say that the priority to this action is the highest. If an action isn't within a nested for loop, then you must say that the priority isn't the highest.
So, following the instructions, can you provide a high-level explanation of the provided program that another instance of LLM can rewrite this program from that summary? Remember to verify the priority of the actions, they should be well explained in your text (e.g., this action has the highest priority, this has the second highest and so on).

B.2.1.2 Reconstructor Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:
     - how long they survive a battle
     - how much damage they can inflict to opponent units
     - how close they need to be from opponent units to attack them.
  3. They can attack the oppponent units.
- Resource units:
  1. The source of resources, doesn't belong to any player.
  2. These units cannot execute any actions.
  3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.

Here, I provided a context free grammar (CFG) of microRTS playing strategy inside the $< CFG >< /CFG >$ tag written bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u$.hasNumberOfUnits(T, N)
| u.opponentHasNumberOfUnits(T, N)
| u.hasLessNumberOfUnits(T, N)
| u.haveQtdUnitsAttacking(N)
| u.hasUnitWithinDistanceFromOpponent(N)
| u.hasNumberOfWorkersHarvesting(N)
| u.canAttack()
| u.hasUnitThatKillsInOneAttack()
| u.opponentHasUnitThatKillsUnitInOneAttack()
| u.hasUnitInOpponentRange()
| u.opponentHasUnitInPlayerRange()
| u.canHarvest()

$C \rightarrow u$.build(T, D, N) | u.train(T, D, N) | u.moveToUnit(T_p, O_p)
| u.attack(O_p) | u.harvest(N)
| u.idle() | u.moveAway()

$T \rightarrow$ Base | Barracks | Ranged | Heavy
| Light | Worker

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest
| LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

$< /CFG >$

This CFG allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units. The following describes a scripting language for playing MicroRTS.
The Boolean functions are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.

2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.

3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.

4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.

5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.

6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.

7. u.is_Type(T): Checks if a unit is an instance of type T.

8. u.isBuilder(): Checks if a unit is of type Worker.

9. u.canAttack(): Checks if a unit can attack.

10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.

11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action. v u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.

12. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.

13. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions are described below. These functions assign actions to units.

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.idle(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base. 'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions. 'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players. 'e' is an empty block which means doing nothing.

The for loops in this scripting language iterate over all units and the instructions inside the for loops attempt to assign actions to each of these units. For example, for (Unit u) u.build(Barracks, EnemyDir, 8)
The snippet above will assign a build action to unit u. Note that the only unit that can build is Worker. In the for loop above, if u is Ranged, for example, then the instruction u.build(Barracks,EnemyDir,8) will be ignored. Once an action is assigned to a unit, it cannot be changed. That is why the for loops offer a priority scheme to the actions. The way that for loop are organized is perhaps the most important feature of this language. The program always has the form: for (Unit u) ... With the possibility of adding nested for-loops that go through all units. The parameter in each instruction limits the amount of the thing that is trained or build. For example, u.build(Barracks, EnemyDir, 8) limits to at most 8 Barracks. If the player already has 8 barracks, then this instruction will be ignored. Now, you have the background you need to know!
Here is a summary of a programmatic strategy generated by a large language model. Let's Call it 'LLM-Explanation' which was generated as an explanation of a strategic program for playing MicroRTS.

Explanataion E

The following 7 are some guidelines for writing a program in MicroRTS:

1. There is NO NEED TO write classes, initiate objects such as Unit, Worker, etc.

2. You should NOT WRITE any comments in the code. (A comment is written in this format //comment, so avoid it!)

3. Use curly braces like C/C++/Java while writing any 'for' or 'if' or 'if-else' block. Start the curly braces in the same line of the block.

4. Do not write 'else if(B) {' block. Write 'else { if(B) {...}}' instead.

5. The format of the generated code must be the same as the example provided earlier. (not the content, the just the general structure)

6. Write only the pseudocode inside '$< strategy >< /strategy >$' tag.

7. The strategy should be written inside one or several 'for' blocks.

Your tasks are the following 11:

1. Understand the meanings of all the boolean (B) and command (C) functions from above and try to relate them in the context of microRTS playing strategies.

2. Given the instructions on how to write a program and the explanation from the large language model('LLM-Explanation') provided earlier, can you write down the strategy encoded in the explanation and reconstruct the program in the MicroRTS scripting language? (Please only use the language provided)

3. You must not use any symbols (for example: &&, ||, etc.) outside this given CFG. You have to strictly follow this CFG while writing the pseudocode.

4. Look carefully, the methods of non-terminal symbols B and C have prefixes 'u.' in the examples since they are methods of the object 'Unit u'. You also need to follow the patterns of the example provided earlier.

5. Write only the pseudocode inside '$< strategy >< /strategy >$' tag.

6. Do not write unnecessary symbols of the CFG such as, '$- >$', '$- >$', etc.

7. When you encounter parentheses in an expression or code, their opening and closing positions are crucial as they indicate the inclusion of some statements within others. The most important feature of this language is how the nested for-loops work and where they start and finish. The for-loops offer an hierarchy that determines the priority of the actions. The list below specifies the different priorities for actions one can obtain with the for-loops (from highest to lowest priority).

8. Actions inserted in nested for-loops at the top of the program receives the highest priority.

9. Actions inserted in nested for-loops that appear later in the program have higher priority than actions that appear outside a nested for loop.

10. Actions outside the nested for-loops that appear earlier in the program have higher priority than actions outside for-loops that appear later in the program.

11. Check the pseudocode and ensure it does not violate the rules of the CFG or the guidelines of writing the strategy.

IMPORTANT:

- Conditional structures such as if-statements are rarely needed. Use an if-statement only if you are sure that you need them to implement the strategy. For example, you should NOT use if-statements to ensure that a number of units is trained as these numbers are already handled by the action functions.
- For efficiency reasons, your program should have at most 2 levels of nested loops.
- Double check whether the priorities of the actions are matching with the nested for-loop structure of your program.
- The instructions with highest priority should be placed in innermost loops. If you aren't sure about an action, then leave it in the main for-loop.

B.2.1.3 Verifier Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:

- how long they survive a battle
- how much damage they can inflict to opponent units
- how close they need to be from opponent units to attack them.

3. They can attack the oppponent units.

- Resource units:

1. The source of resources, doesn't belong to any player.
2. These units cannot execute any actions.
3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.
Here, I provided a context free grammar (CFG) of microRTS playing strategy inside the $< CFG >< /CFG >$ tag written bellow:

$< CFG >$

$S \rightarrow SS$ | for(Unit u) S | if(B) then S
| if(B) then S else S | C | λ

$B \rightarrow u$.hasNumberOfUnits(T, N)
| u.opponentHasNumberOfUnits(T, N)
| u.hasLessNumberOfUnits(T, N)
| u.haveQtdUnitsAttacking(N)
| u.hasUnitWithinDistanceFromOpponent(N)
| u.hasNumberOfWorkersHarvesting(N)
| u.canAttack()
| u.hasUnitThatKillsInOneAttack()
| u.opponentHasUnitThatKillsUnitInOneAttack()
| u.hasUnitInOpponentRange()
| u.opponentHasUnitInPlayerRange()
| u.canHarvest()

$C \rightarrow u$.build(T, D, N) | u.train(T, D, N) | u.moveToUnit(T_p, O_p)
| u.attack(O_p) | u.harvest(N)
| u.idle() | u.moveAway()

$T \rightarrow$ Base | Barracks | Ranged | Heavy
| Light | Worker

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
| 10 | 15 | 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest
| LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

$< /CFG >$

This CFG allows nested loops and conditionals. It contains several Boolean functions (B) and command-oriented functions (C) that provide either information about the current state of the game or commands for the ally units. The following describes a scripting language for playing MicroRTS.
The Boolean functions are described below:

1. u.hasNumberOfUnits(T, N): Checks if the ally player has N units of type T.
2. u.opponentHasNumberOfUnits(T, N): Checks if the opponent player has N units of type T.

3. u.hasLessNumberOfUnits(T, N): Checks if the ally player has less than N units of type T.

4. u.haveQtdUnitsAttacking(N): Checks if the ally player has N units attacking the opponent.

5. u.hasUnitWithinDistanceFromOpponent(N): Checks if the ally player has a unit within a distance N from a opponent's unit.

6. u.hasNumberOfWorkersHarvesting(N): Checks if the ally player has N units of type Worker harvesting resources.

7. u.is_Type(T): Checks if a unit is an instance of type T.

8. u.isBuilder(): Checks if a unit is of type Worker.

9. u.canAttack(): Checks if a unit can attack.

10. u.hasUnitThatKillsInOneAttack(): Checks if the ally player has a unit that kills an opponent's unit with one attack action.

11. u.opponentHasUnitThatKillsUnitInOneAttack(): Checks if the opponent player has a unit that kills an ally's unit with one attack action. v u.hasUnitInOpponentRange(): Checks if an unit of the ally player is within attack range of an opponent's unit.

12. u.opponentHasUnitInPlayerRange(): Checks if an unit of the opponent player is within attack range of an ally's unit.

13. u.canHarvest(): Checks if a unit can harvest resources.

The Command functions are described below. These functions assign actions to units.

1. u.build(T, D, N): Builds N units of type T on a cell located on the D direction of the unit.

2. u.train(T, D, N): Trains N units of type T on a cell located on the D direction of the structure responsible for training them.

3. u.moveToUnit(T_p, O_p): Commands a unit to move towards the player T_p following a criterion O_p.

4. u.attack(O_p): Sends N Worker units to harvest resources.

5. u.harvest(N): Sends N Worker units to harvest resources.

6. u.idle(): Commands a unit to stay idle and attack if an opponent unit comes within its attack range.

7. u.moveAway(): Commands a unit to move in the opposite direction of the player's base. 'T' represents the types a unit can assume. 'N' is a set of integers. 'D' represents the directions available used in action functions. 'O_p' is a set of criteria to select an opponent unit based on their current state. 'T_p' represents the set of target players. 'e' is an empty block which means doing nothing.

The for loops in this scripting language iterate over all units and the instructions inside the for loops attempt to assign actions to each of these units. For example, for (Unit u) u.build(Barracks, EnemyDir, 8)
The snippet above will assign a build action to unit u. Note that the only unit that can build is Worker. In the for loop above, if u is Ranged, for example, then the instruction u.build(Barracks,EnemyDir,8) will be ignored. Once an action is assigned to a unit, it cannot be changed. That is why the for loops offer a priority scheme to the actions. The way that for loop are organized is perhaps the most important feature of this language. The program always has the form: for (Unit u) ... With the possibility of adding nested for-loops that go through all units. The parameter in each instruction limits the amount of the thing that is trained or build. For example, u.build(Barracks, EnemyDir, 8) limits to at most 8 Barracks. If the player already has 8 barracks, then this instruction will be ignored. Now, you have the background you need to know!

I have a program P which is written in MicroRTS language. It provides an strategy for playing the real-time strategy game. Here is the program:
Program P

Then, I'm asking the first instance of an LLM for a high-level explanation without any programming jargon about the program so that another instance of LLM can regenerate the code using that explanation. The first LLM gave me this:

Explanation E

Now, I want to know if the explanation is using any computer programming jargon or provides step by step or line by line instructions to reconstruct the program? (Answer with yes or no first and then explain why)

B.2.2 Human-crafted Set

B.2.2.1 Explainer Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:
     - how long they survive a battle
     - how much damage they can inflict to opponent units
     - how close they need to be from opponent units to attack them.
  3. They can attack the oppponent units.
- Resource units:
  1. The source of resources, doesn't belong to any player.
  2. These units cannot execute any actions.
  3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.

The code for playing this game is written in Java. Here is a list of classes, functions, and attributes that you need to be familiar with:

- GameState Class functions:
    1. gs: is the a gamestate object which contains all the information about the state of the game in the MicroRTs game. (gs is the notation that will be observed a lot)
    2. getPhysicalGameState(): Returns the physical game state (the actual 'map') of a microRTS game associated with this state.
    3. getPlayer(int playerID): given a player ID, returns the player object.
    4. getActionAssignment(Unit u): It returns the action assigned to a unit.
    5. getResourceUsage(): It gets the resource usage for a gamestate. It also has another function inside which is getResourcesUsed(player). If you want to see how much resource is used for a player, you can use: resourcesUsed=gs.getResourceUsage().getResourcesUsed(player).
- Unit Class functions:
    1. getType(): Returns the type of the unit. If we want to check the unit type, we should call this (e.g if u.getType()==...)
    2. getX(): returns the x position of the unit on the board.
    3. getY(): returns the y position of the unit on the board.
    4. getPlayer(): returns the ID of the unit owner. The unit ownership in the game is identified by specific IDs. Player1 is assigned the ID of 0, while Player2 is given the ID of 2. Resources in the game are neutral and do not belong to any side, so they are assigned a unit owner ID of -1, indicating that they are unowned by any player.
- Player Class functions:
    1. getID(): Returns the player ID.
    2. getResources(): Returns the amount of resources owned by the player.
- Harvest Class function:
    1. This class is extending the AbstractAction class.
    2. getTarget(): Returns the target for the harvest action.
    3. getBase(): Returns the base for the harvest action.
- UnitType Class attributes:

1. cost: Cost to produce a unit of a type. It's an attribute of the Type class, not a function. (e.g -¿ workerType.cost)
2. isResource: Boolean that returns True if a unit is of type "resource". This is used to check if the type of a unit is Resource or not.
3. canHarvest: Boolean that returns True if a unit can harvest resources (it should be a worker)
4. isStockpile :Boolean that returns True if resources can be returned to this unit type.
5. canAttack: Boolean that returns True if a unit of this type can attack.
6. canMove: Boolean that returns True if a unit of this type can move.

- Different UnitTypes:

1. workerType
2. baseType
3. barracksType
4. lightType
5. heavyType
6. rangedType

- Other Functions:

1. train(u, type): command unit u to train a unit of a type. Note that base units can train workers and barrack units can train combat units (light, heavy, ranged)
2. attack(u, targetUnit): command unit u to attack the target unit. (if you need to say attack no where, you can use attack(u, null)).
3. buildIfNotAlreadyBuilding(u, type, u.getX(), u.getY(), reservedPositions ,p, pgs): it calls the build action after finding the position that the new unit should be built. You don't need to worry about the reversedPositions variable. It will be defined as an empty list of integar. (You can define it as: 'List¡Integer¿ reservedPositions = new LinkedList¡¿()' before using it)
4. harvest(u, targetUnit, baseUnit): command unit to harvest from the resources and add to the base unit.
5. heavyType
6. rangedType

There are so many classes in this framework implementation, for instance, Player, GameState, PhysicalGameState, etc are all classes as mentioned above. You can also use libraries in python such as Math for calculation.

Now, given the above information, try to understand and relate the classes, functions, and attributes explained above in the context of microRTS playing strategies.

Alright, let's take a look at program P in which I want you to write an explanation for:

Program P

The following 9 are some guidelines for writing an explanation for this strategy:

1. Please write a high-level explanation and do not explain the code line by line but try to include numbers and unit names in your natural language explanation.

2. Try to understand what is happening in the code and explain it in natural language for someone who wants to know how to play MicroRTS using this strategy.

3. Write the explanation inside '¡explanation¿¡/explanation¿' tag.

4. DON'T USE any quotation marks in writing the explanation.

5. At the end of the explanation, write the overall goal of the strategy as well. (include it inside the explanation tag)

6. Don't forget to mention the numbers but in a natural language way.

7. If there is a formula to calculate something, explain it in natural language as well.

8. Pay attention to the nested 'for-loops' and nested 'if statements' becuase they determine the prority of action being called. Try to reflect this order in your explanation.

9. You MUST NOT talk in terms of for-loops and programming language jargon as people not familiar with programming will not understand the explanation. Also, you MUST not use any 'nest' or 'nested' words as they are related to programming elements. You should talk in terms of priorities of the actions, as you would explain the strategy to a gamer.

So, following the instructions, can you provide a high-level explanation of the provided program that another instance of LLM can rewrite this program from that summary?

B.2.2.2 Reconstructor Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:
     - how long they survive a battle
     - how much damage they can inflict to opponent units

- how close they need to be from opponent units to attack them.

3. They can attack the oppponent units.

- Resource units:

1. The source of resources, doesn't belong to any player.
2. These units cannot execute any actions.
3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.

The code for playing this game is written in Java. Here is a list of classes, functions, and attributes that you need to be familiar with:

- GameState Class functions:

1. gs: is the a gamestate object which contains all the information about the state of the game in the MicroRTs game. (gs is the notation that will be observed a lot)
2. getPhysicalGameState(): Returns the physical game state (the actual 'map') of a microRTS game associated with this state.
3. getPlayer(int playerID): given a player ID, returns the player object.
4. getActionAssignment(Unit u): It returns the action assigned to a unit.
5. getResourceUsage(): It gets the resource usage for a gamestate. It also has another function inside which is getResourcesUsed(player). If you want to see how much resource is used for a player, you can use: resourcesUsed=gs.getResourceUsage().getResourcesUsed(player).

- Unit Class functions:

1. getType(): Returns the type of the unit. If we want to check the unit type, we should call this (e.g if u.getType()==...)
2. getX(): returns the x position of the unit on the board.
3. getY(): returns the y position of the unit on the board.

4. getPlayer(): returns the ID of the unit owner. The unit ownership in the game is identified by specific IDs. Player1 is assigned the ID of 0, while Player2 is given the ID of 2. Resources in the game are neutral and do not belong to any side, so they are assigned a unit owner ID of -1, indicating that they are unowned by any player.

- Player Class functions:
  1. getID(): Returns the player ID.
  2. getResources(): Returns the amount of resources owned by the player.
- Harvest Class function:
  1. This class is extending the AbstractAction class.
  2. getTarget(): Returns the target for the harvest action.
  3. getBase(): Returns the base for the harvest action.
- UnitType Class attributes:
  1. cost: Cost to produce a unit of a type. It's an attribute of the Type class, not a function. (e.g -¿ workerType.cost)
  2. isResource: Boolean that returns True if a unit is of type "resource". This is used to check if the type of a unit is Resource or not.
  3. canHarvest: Boolean that returns True if a unit can harvest resources (it should be a worker)
  4. isStockpile :Boolean that returns True if resources can be returned to this unit type.
  5. canAttack: Boolean that returns True if a unit of this type can attack.
  6. canMove: Boolean that returns True if a unit of this type can move.
- Different UnitTypes:
  1. workerType
  2. baseType
  3. barracksType
  4. lightType
  5. heavyType
  6. rangedType

- Other Functions:
    1. train(u, type): command unit u to train a unit of a type. Note that base units can train workers and barrack units can train combat units (light, heavy, ranged)
    2. attack(u, targetUnit): command unit u to attack the target unit. (if you need to say attack no where, you can use attack(u, null)).
    3. buildIfNotAlreadyBuilding(u, type, u.getX(), u.getY(), reservedPositions ,p, pgs): it calls the build action after finding the position that the new unit should be built. You don't need to worry about the reversedPositions variable. It will be defined as an empty list of integar. (You can define it as: 'List¡Integer¿ reservedPositions = new LinkedList¡¿()' before using it)
    4. harvest(u, targetUnit, baseUnit): command unit to harvest from the resources and add to the base unit.
    5. heavyType
    6. rangedType

There are so many classes in this framework implementation, for instance, Player, GameState, PhysicalGameState, etc are all classes as mentioned above. You can also use libraries in python such as Math for calculation.

Now, you have the background you need to know!
Now, given the above information, try to understand and relate the classes, functions, and attributes explained above in the context of microRTS playing strategies.

Here is a summary of a programmatic strategy generated by a large language model. Let's Call it 'LLM-Explanation' which was generated as an explanation of a strategic program for playing MicroRTS.

Explanation E

The following 7 are some guidelines for writing a program in MicroRTS:

1. There is NO NEED TO write classes, initiate objects such as Unit, Worker, etc.

2. Write all the code in one section. Don't write any functions. But don't leave any section non-implemented.

3. Write only the pseudocode inside '¡strategy¿¡/strategy¿' tag.

4. Suppose gs(GameState object), pgs(PhysicalGameState object), p(Player object), and player(player ID, int) are defined and given. Use the same syntax in your code. You don't need to define them beforehand.

5. Don't use any external information from any resources. Just rely on the explanation and the prompt provided to generate the code.

6. Only write the strategy code, I don't need any explanations.

7. Once again, remember to implement everything. Don't leave any helper function unimplemented.

Your tasks are the following 5:

1. Understand the meaning of all functions and variables and try to relate them in the context of microRTS playing strategies.

2. Given the instructions on how to write a program and the explanation from the large language model('LLM-Explanation') provided earlier, can you write down the strategy encoded in the explanation and reconstruct the program in the MicroRTS scripting language? (Please only write in Java)

3. Make sure you only use the functions, and attributes that I introduced earlier. For instance, don't call an object of a class with the function of the other class. (e.g we don't have gs.getID())

4. Check the generated code and make sure you are following Java syntax. The code needs to be ready to get compiled right away.

5. Don't use any assumptions of your own except for the ones that I noted to write the code.

B.2.2.3 Verfier Prompt

MicroRTS (ONTANÓN, 2013) is an implementation of a real-time strategy game, played between two players. Each player controls a set of units of different types.

- Worker units can:
  1. collect resources
  2. build structures (Barracks and Bases)
  3. attack opponent units
- Barracks and Bases:
  1. Barracks can train combat units. (they can produce Light, Heavy or Ranged units)
  2. Bases can train the Workers. (they can only produce Workers)
  3. They can neither attack opponents units nor move.
- Combat units:
  1. can be of type Light, Heavy, or Ranged.
  2. These units differ in:
     - how long they survive a battle
     - how much damage they can inflict to opponent units
     - how close they need to be from opponent units to attack them.
  3. They can attack the oppponent units.
- Resource units:
  1. The source of resources, doesn't belong to any player.
  2. These units cannot execute any actions.
  3. When the number of resources left reaches 0, the unit disappears. (It only has one parameter: resources left.)

Actions are deterministic and there is no hidden information in MicroRTS. A match is played on a map and each map might require a different strategy for defeating the opponent.

The code for playing this game is written in Java. Here is a list of classes, functions, and attributes that you need to be familiar with:

- GameState Class functions:

1. gs: is the a gamestate object which contains all the information about the state of the game in the MicroRTs game. (gs is the notation that will be observed a lot)
2. getPhysicalGameState(): Returns the physical game state (the actual 'map') of a microRTS game associated with this state.
3. getPlayer(int playerID): given a player ID, returns the player object.
4. getActionAssignment(Unit u): It returns the action assigned to a unit.
5. getResourceUsage(): It gets the resource usage for a gamestate. It also has another function inside which is getResourcesUsed(player). If you want to see how much resource is used for a player, you can use: resourcesUsed=gs.getResourceUsage().getResourcesUsed(player).

- Unit Class functions:

1. getType(): Returns the type of the unit. If we want to check the unit type, we should call this (e.g if u.getType()==...)
2. getX(): returns the x position of the unit on the board.
3. getY(): returns the y position of the unit on the board.
4. getPlayer(): returns the ID of the unit owner. The unit ownership in the game is identified by specific IDs. Player1 is assigned the ID of 0, while Player2 is given the ID of 2. Resources in the game are neutral and do not belong to any side, so they are assigned a unit owner ID of -1, indicating that they are unowned by any player.

- Player Class functions:

1. getID(): Returns the player ID.
2. getResources(): Returns the amount of resources owned by the player.

- Harvest Class function:

1. This class is extending the AbstractAction class.
2. getTarget(): Returns the target for the harvest action.
3. getBase(): Returns the base for the harvest action.

- UnitType Class attributes:

1. cost: Cost to produce a unit of a type. It's an attribute of the Type class, not a function. (e.g -¿ workerType.cost)

2. isResource: Boolean that returns True if a unit is of type "resource". This is used to check if the type of a unit is Resource or not.
3. canHarvest: Boolean that returns True if a unit can harvest resources (it should be a worker)
4. isStockpile :Boolean that returns True if resources can be returned to this unit type.
5. canAttack: Boolean that returns True if a unit of this type can attack.
6. canMove: Boolean that returns True if a unit of this type can move.

- Different UnitTypes:

1. workerType
2. baseType
3. barracksType
4. lightType
5. heavyType
6. rangedType

- Other Functions:

1. train(u, type): command unit u to train a unit of a type. Note that base units can train workers and barrack units can train combat units (light, heavy, ranged)
2. attack(u, targetUnit): command unit u to attack the target unit. (if you need to say attack no where, you can use attack(u, null)).
3. buildIfNotAlreadyBuilding(u, type, u.getX(), u.getY(), reservedPositions ,p, pgs): it calls the build action after finding the position that the new unit should be built. You don't need to worry about the reversedPositions variable. It will be defined as an empty list of integar. (You can define it as: 'List¡Integer¿ reservedPositions = new LinkedList¡¿()' before using it)
4. harvest(u, targetUnit, baseUnit): command unit to harvest from the resources and add to the base unit.
5. heavyType
6. rangedType

There are so many classes in this framework implementation, for instance, Player, GameState, PhysicalGameState, etc are all classes as mentioned

above. You can also use libraries in python such as Math for calculation.

Now, you have the background you need to know!
Then, given the above information, try to understand and relate the classes, functions, and attributes explained above in the context of microRTS playing strategies.

I have a program P which is written in Java language as a part of strategy for playing MicroRTS. Here is the program:
Program P

Then, I'm asking the first instance of an LLM for a high-level explanation without any programming jargon about the program so that another instance of LLM can regenerate the code using that explanation. The first LLM gave me this:

Explanation E

Now, I want to know if the explanation is using any computer programming jargon or provides step by step or line by line instructions to reconstruct the program? (Answer with yes or no first and then explain why)

B.3 Computer Programming Prompts

B.3.1 Explainer Prompt

Can you explain in plain English, without programming jargon, and without saying what the algorithm does (that is you cannot just say it *does something (e.g. computes the proper subset of a set of arguments)*, what this program does? It needs to be detailed enough so that another LLM is able to write this program from your description.
Be careful to give me a high-level natural language explanation. I don't want to see any variable names and code elements in the text.
Also, write the explanation inside '<explanation></explanation>' tag.
REMEMBER, don't use any programming jargon!

B.3.2 Reconstructor Prompt

> Can you write the encrypted strategy in the explanation below in C?
>
> *Explanation E*

B.3.3 Verifier Prompt

> I have a program P which that *does something*. Here is the program:
>
> *Program P*
>
> Then, I'm asking an LLM for a high-level explanation of the program without any programming jargon so that another LLM can regenerate the code using that explanation.
> The first LLM gave me this:
>
> *Explanation E*
>
> Now, I want to know if the explanation uses any computer programming jargon or provides step by step or line by line instructions to reconstruct the program? (Answer with yes or no first and then explain why)

B.3.3.1 Verifier Examples

Here are two explanations for Program 2, produced by the LLM from the pool of C programming problems that received distinct feedback from the Verifier.

P2:

```
#include <stdio.h>
#include <stdlib.h>

#define _ F-->00 || F-OO--;

long F=00,OO=00;

F_OO()
```

```
{
            _-_-_-_
        _-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_
   _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
   _-_-_-_-_-_-_-_-_-_-_-_-_-_
     _-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_
            _-_-_-_
}
main(){F_OO();printf("%1.3f\n", 4.*-F/OO/OO);}
```

Explanataion 1: The program consists of two main parts. The first part sets up two long numbers with a value of zero. It then defines a pattern using the two numbers in such a way that their values change according to the defined pattern. The pattern looks like a triangle when visualized, starting with a single instance at the top, expanding in width as it goes down, and then contracting again towards the bottom. The process involves decreasing the first number if certain conditions involving the two numbers are met, while the second number keeps increasing. This operation goes on until the first number reaches a certain condition.

The second part of the program uses the values of the two numbers after the pattern operation and performs a calculation using them. This calculation involves multiplying the negative of the first number by 4 and then dividing by the square of the second number. The result of this calculation is then displayed with three decimal places.

Verfier Response for Explanation 1: Yes.
The provided explanation contains programming jargon such as "long numbers," indicating data types, and references conditional and iterative operations through phrases like "certain conditions" and "keeps increasing." Furthermore, specific mathematical operations are detailed, and the triangular pattern visualization implies a specific code structure. These elements, together, give insights into the logic and operations of the program, potentially aiding in its reconstruction.

Explanataion 2: Imagine you have two containers. One container is filled up to a certain level, and the second one is empty. You start a process in which you keep taking a little bit from the filled container and putting it into the empty container. The process has two specific rules: 1. You only stop transferring when the filled container is empty. 2. The rate at which you take from the filled container decreases every time.
After the process is finished, you examine the amount you've moved compared to the original amount in the filled container. Then, using this relationship, you calculate a number and display it.

Verfier Response for Explanation 2: No.
The explanation does not use computer programming jargon, and it doesn't provide step-by-step instructions to reconstruct the program. The given description uses an analogy of two containers and a transfer process between them, without referencing any specific programming constructs or operations. It describes the behavior of the program at a high level, but it does not give details about how the program implements that behavior.

B.4 C Programs Pool

In this section, we present the 10 C programs used in our experiments. The first subsection features the obfuscated programs, accompanied by their explanations and one sample of our k attempts at reconstructing the program. As noted in the paper, none of these attempts successfully reconstructed the program. Following that, the subsequent subsection shows the non-obfuscated implementations alongside one sample from our k successful reconstructions, reaffirming that each was accurately reconstructed as mentioned in the paper.

B.4.1 Obfuscated C Programs

P1: This program computes all proper subsets of the set of arguments passed to it.

```
#include <stdio.h>
#include <stdlib.h>

main(Q,O)char**O;{
    if(--Q){
        main(Q,O);
        O[Q][0]^=0X80;
        for(O[0][0]=0;O[++O[0][0]]!=0;)
            if(O[O[0][0]][0]>0)puts(O[O[0][0]]);
        puts("----------");
        main(Q,O);
    }
}
```

Explanation:
Imagine you have a collection of phrases. The program performs a unique action on this collection: Starting from the last phrase and moving towards the first, for each phrase, it: Repeats the entire routine with the preceding phrases. Alters the start of the current phrase in a special way. Goes through the collection from the start. For every phrase that begins in a certain manner, it shows it. Displays a line of dashes. Once more, carries out the entire routine with the preceding phrases. It's like a set of instructions that continuously loops for each phrase, making a specific change and then showcasing certain phrases based on a particular feature of their beginning
Reconstructed Program for P1:

```
#include <stdio.h>
#include <string.h>

#define MAX_PHRASES 10
#define MAX_LENGTH 100

char phrases[MAX_PHRASES][MAX_LENGTH];

void alter_phrase(char *phrase) {
    // Example alteration: prepend "* " to the phrase.
    char temp[MAX_LENGTH];
    strcpy(temp, "* ");
    strcat(temp, phrase);
    strcpy(phrase, temp);
}
```

```
int begins_in_certain_manner(const char *phrase) {
    // Example check: phrase begins with "* "
    return strncmp(phrase, "* ", 2) == 0;
}

void perform_routine(int index) {
    if (index < 0) return;
    perform_routine(index - 1);
    alter_phrase(phrases[index]);
    for (int i = 0; i < MAX_PHRASES; i++) {
        if (begins_in_certain_manner(phrases[i])) {
            printf("%s\n", phrases[i]);
        }
    }
    printf("-----------------\n");
    perform_routine(index - 1);
}

int main() {
    strcpy(phrases[0], "Hello");
    strcpy(phrases[1], "World");
    strcpy(phrases[2], "How");
    strcpy(phrases[3], "Are");
    strcpy(phrases[4], "You");
    perform_routine(4);
    return 0;
}
```

P2: This program calculates the value of π.

```
#include <stdio.h>
#include <stdlib.h>

#define \_ F-->00 || F-OO--;

long F=00,OO=00;

F\_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
```

```
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
}
main(){F_OO();printf("%1.3f\n", 4.*-F/OO/OO);}
```

Explanation:
Imagine a bookkeeper with two tallies, both starting at zero. Now, there's a curious poem-like structure which, when read, provides a series of actions to follow. This poem is a mix of lines, with some lines repeated more times than others. Every time a certain symbol is encountered in the poem, the bookkeeper has to do two things:

Check if the first tally has reached a certain condition; if not, decrease it. If the first tally has reached that condition, then check if the second tally meets another condition; if not, decrease it.
After the bookkeeper completes reading through this poem and performs all the actions, there's a simple arithmetic task to be done. Multiply the negative value of the first tally by four, then divide it by the square of the second tally. The result of this arithmetic operation is then shared.
In essence, the poem drives the two tallies towards certain values, and then an arithmetic operation is done using these tallies to give a final result.

Reconstructed Program for P2:

```
#include <stdio.h>
#include <string.h>

// Function to read the poem and adjust the tallies
void readPoemAndAdjustTallies(const char *poem,\
    int *tally1, int *tally2) {
    char symbol = '#';
    for (int i = 0; i < strlen(poem); i++) {
        if (poem[i] == symbol) {
```

```
            if (*tally1 > -10) {
                (*tally1)--;
            } else if (*tally2 > -5) {   necessary
                (*tally2)--;
            }
        }
    }
}
int main() {
    char poem[] = "Roses are red,\n"
                  "#Violets are blue,\n"
                  "Every time I see a #,\n"
                  "I think of you.\n"
                  "#\n"
                  "Books and poems,\n"
                  "#Symbols and signs,\n"
                  "Every # is a moment,\n"
                  "That intertwines.\n";

    int tally1 = 0, tally2 = 0;
    readPoemAndAdjustTallies(poem, &tally1, &tally2);
    if (tally2 == 0) {
        printf("Cannot perform the arithmetic\
        operation since the second tally is zero.\n");
        return 1;
    }
    double result = (-4.0 * tally1) / (tally2 * tally2);
    printf("The result is: %lf\n", result);
    return 0;
}
```

P3: This program prints the input string sorted alphabetically.

```
#include "stdio.h"
main(argc, argv)
int argc;
char **argv;
{
while (*argv != argv[1] && (*argv = argv[1]) && (argc = 0) ||
(*++argv && (**argv && ((++argc)[*argv] && (**argv <= argc[*argv]
||(**argv += argc[*argv] -= **argv = argc[*argv] - **argv)) &&
--argv || putchar(**argv) && +++argv--) || putchar(10))));
}
```

Explanation:
Imagine you have a line of people holding cards with numbers, and next to them is another line of people holding cards with letters. We're focusing on the people holding letters, but they'll sometimes peek at the numbers from the first line.
For every person in the letter line:
Check the position of this person in their line and look at the number they're holding.
If their position in the line is greater than the number on their card: Take the difference by subtracting the number from their position. Now, add this difference to the number on their card. If their position in the line is less than or equal to the number: Now, take the difference by subtracting their position from the number. Add this difference to the number on their card. After doing this, show the letter from that person's card to the audience.
Once every person in the letter line has shown their card after the above adjustments, there's a big applause (or a break).

However, if there's no one in the letter line, just proceed with the applause (or break).

Reconstructed Program for P3:

```
#include <stdio.h>
int main() {
int n; // Number of people in the letter line

printf("Enter the number of people in the letter line: ");
scanf("%d", &n);

if (n == 0) {
    printf("Applause (or break)!\n");
    return 0;
}

int numbers[n];
char letters[n];
printf("Enter the numbers and letters for each person:\n");
for (int i = 0; i < n; i++) {
    printf("Person %d - Number: ", i+1);
    scanf("%d", &numbers[i]);

    printf("Person %d - Letter: ", i+1);
    scanf(" %c", &letters[i]);
}
```

```
for (int i = 0; i < n; i++) {
    int position = i + 1;
    if (position > numbers[i]) {
        int difference = position - numbers[i];
        numbers[i] += difference;
    } else {
        int difference = numbers[i] - position;
        numbers[i] += difference;
    }

    printf("Showing letter: %c\n", letters[i]);
}

printf("Applause (or break)!\n");

return 0;
}
```

P4: This program prints the first 100 prime numbers.

```
#include <stdio.h>
#define  f(f,g){z e=0;for(;e<f;e++)g;}
#define  i(f,g)static z f(z a){return g;}
#define  j(f,g)static void f(z*a,z*b,z*c){g}
#define  h(f,g)static z f(z a,z b,z c){return g;}
#define  g(f,g,h,i,j)static z f(z b){z a=g,c=h;for(;i)a=j;return a;}
typedef unsigned char y;typedef unsigned long long z;extern y*w;
static z b(z a,z b){return a>>b|a<<(64-b);}i(_,(a>>6)^b(a,61)
^b(a,19))i(_a,b(a,39)^b(a,28)^b(a,34))h(x,((a^b)&c)^(a&b))
i(u,b(a,41)^b(a,18)^b(a,14))h(t,(((((3*(a*c+b*b)>>9)+(3*b*c>>32))
*a>>21)+(3*a*a*b>>6)+((b>>4)*(b>>4)*b>>46))>>18)+a*a*a)
h(m,t((b<<16)|(c>>48),(c>>24)%(1<<24),
c%(1<<24))>>48<a)h(s,(a&b)^(~a&c))i(r,b(a,1)^b(a,8)^(a>>7))
g(o,0,0,c<8;c++,a*256+w[b*8+c])g(d,0,0,c<13;c++,a*31+w[b*13+c]-96)
g(p,0,4,c;c/=2,a|c*m(b,a|c,a)
)g(q,0,1ull<<63,c;c/=2,a|c*m(b,p(b),a|c))g(v,b>1,2,c<b;c++,a&&b%c)
g(l,b?l(b-1)+1:2,a,!v(c);c++,c+1)j(n,z d=a[7]+u(a[4])+s(a[4],a
[5],a[6])+q(l(*b))+c[*b%16];f(8,a[7-e]=e-3?e-7?a[6-e]:d+_a(a[0])+
x(a[1],a[2],a[3]):d+a[3])f(16*(*b%16>14),
c[e]+=c[(e+9)%16]+r(c[(e+1)%16])+_(c[(e+14)%16])))
j(k,f(8,b[e]=a[e])f(80,n(a,&e,c))f(8,a[e]+=b[e]))
int main(){z a[8],b[8],c[16];f(8,a[e]=d(e))
```

```
f(16,c[e]=e-15?o(e):d(8))k(a,b,c);
f(16,c[e]=e?e-15?0:11264:1ull<<63)k(a,b,c);
f(8,printf("%016llx%s",a[e],e-7?"":"\n"))
return!w;}y*w=(y*) "crsmyiajqhwy{unwa|
hjoi‘hlxhpxrzb˜edko˜rtr˜ileqyjk‘znqgsuitvgqnfdfa||
wedvnmhozkpokootqzcexeld˜oibqzpcsuw{ib{x‘m‘hsa‘jmn}wcfzpb";
```

Explanation:
Imagine you have a secret vault with 8 chambers. Initially, each chamber has a unique piece of information. Now, alongside this vault, you also have a special set of 16 cards. Some cards come with information pre-written on them, while others are blank. Now, you perform a series of operations:
You make a duplicate vault and keep it next to the original one. Using the original vault's information and the special set of cards, you adjust the information in the duplicate vault. You do this 80 times. After you're done, you take the adjusted values from the duplicate vault and add them back to the original one. Next, you modify your set of 16 cards. Some of them are reset to blank, and one specific card is marked with a very large number.
You repeat the same series of operations as before:
Make a duplicate of the original vault. Use the original vault and the modified set of cards to adjust the duplicate vault. Merge the adjusted values from the duplicate vault back into the original one. In the end, you read the information from each chamber of the vault and display it.

Reconstructed Program for P4:

```
#include <stdio.h>

#define VAULT_SIZE 8
#define CARD_SET_SIZE 16
#define ADJUSTMENT_TIMES 80

void adjustDuplicateVault(int original[], int cards[],\
    int duplicate[]) {
    for (int i = 0; i < VAULT_SIZE; i++) {
        duplicate[i] = original[i] + cards[i];
    }
}

void mergeDuplicateBack(int original[], int duplicate[]) {
    for (int i = 0; i < VAULT_SIZE; i++) {
        original[i] += duplicate[i];
    }
}
```

```
int main() {
    int vault[VAULT_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
    int cards[CARD_SET_SIZE] = {10, 20, 30, 40, 50, 60,\
                        70, 80, 0, 0, 0, 0, 0, 0, 0, 0};

    // First series of operations
    for (int i = 0; i < ADJUSTMENT_TIMES; i++) {
        int duplicateVault[VAULT_SIZE];
        adjustDuplicateVault(vault, cards, duplicateVault);
        mergeDuplicateBack(vault, duplicateVault);
    }

    // Modify the set of 16 cards
    for (int i = 0; i < CARD_SET_SIZE; i++) {
        cards[i] = 0; // Reset to blank
    }
    cards[0] = 1000; // Mark one card with a very large number

    // Second series of operations
    for (int i = 0; i < ADJUSTMENT_TIMES; i++) {
        int duplicateVault[VAULT_SIZE];
        adjustDuplicateVault(vault, cards, duplicateVault);
        mergeDuplicateBack(vault, duplicateVault);
    }

    // Display the information from each chamber of the vault
    printf("Final vault information:\n");
    for (int i = 0; i < VAULT_SIZE; i++) {
        printf("Chamber %d: %d\n", i + 1, vault[i]);
    }

    return 0;
}
```

P5: This program computes the factorial of a number.

```
#include <stdio.h>
#include <stdlib.h>

#define l11l 0xFFFF
#define ll1 for
#define ll111 if
#define l1l1 unsigned
```

```
#define l111 struct
#define llll1 short
#define ll11l long
#define ll1ll putchar
#define l1111(l) l=malloc(sizeof(l111 llll1));l->llll1=1-1;
l->ll1l1=1-1;
#define l1ll1 *lllll++=l1ll%10000;l1ll/=10000;
#define l1lll ll111(!l1->ll11l){l1111(l1->ll11l);
l1->ll11l->ll1l1=l1;}\
lllll=(l1=l1->ll11l)->lll;ll=1-1;
#define llll 1000

l111 llll1 {
    l111 llll1 *ll11l,*ll1l1;
    l1l1 lll11 lll[llll];
};

int main(){
    l111 llll1 *ll11,*l1l,*l1, *ll1l;
    l1l1 ll11l l1ll;
    ll11l l11,ll,l;
    l1l1 lll11 *llll,*lllll;

    ll1(l=1-1 ;l< 14; ll1ll("\t\"8)>l\"9!.)>vl"[l]^'L'),++l);
    scanf("%d",&l);
    l1l1l(l1l)
    l1l1l(ll11)
    (l1=l1l)->lll[l1l->lll[1-1]=1]=l11l;
    ll1(l11=1+1;l11<=l;++l11){
        l1=ll11;
        llll = (ll1l=(ll11=l1l))->lll;
        lllll=(l1l=l1)->lll;
        ll=(l1ll=1-1);
        ll1(;ll1l->ll1l1||l11l!=*llll;){
            l1ll+=l11**llll++;
            l1ll1 ll111 (++ll>llll){
                l1lll
                llll=(ll1l =l11l->ll1l1)->lll;
            }
        }
        ll1(;l1ll;){
            l1ll1 ll111 (++ll>=llll) {
                l1lll
```

```
            }
        }
        *llllll=l111l;
    }
    ll1(l=(ll=1-1);(l<llll)&&(l1->lll[l]!=l111l);++l);
    ll1(;l1;l1=l1->l1111,l=llll){
        ll1(--l;l>=1-1;--l,++ll)
        printf((ll)?((ll%19)?"%04d":(ll=19,"\n%04d")):\
        "%4d",l1->lll[l]);
    }
    l1111(10);
    return 0;
}
```

Explanation:

This program starts by showing a sequence of symbols on the screen. After that, it waits for you to give it a number. Using this number, the program goes through a series of steps, repeating them as many times as the number you provided.
It keeps the outcomes of these steps in a series of containers. If one container becomes full, it moves on to fill the next one.
After finishing all the steps, the program shows you the outcomes. It starts by skipping some numbers in the first container that are the same as a certain value. Then, it shows you numbers from each container, grouping them in sets. After showing a certain number of sets, it moves to the next line and keeps showing the numbers. This continues until it has shown you everything. At the end, it shows a little gap and then stops.

Reconstructed Program for P5:

```
#include <stdio.h>

#define CONTAINER_SIZE 10
#define TOTAL_CONTAINERS 5
#define SKIP_VALUE -1
#define SET_SIZE 3
int containers[TOTAL_CONTAINERS][CONTAINER_SIZE];
void performSteps(int number) {
    int currentContainer = 0;
    int currentPosition = 0;
    for (int i = 0; i < number; i++) {
        if (currentPosition >= CONTAINER_SIZE) {
            currentPosition = 0;
```

```
            currentContainer++;
            if (currentContainer >= TOTAL_CONTAINERS) {
                printf("All containers are full!\n");
                return;
            }
        }
        containers[currentContainer][currentPosition] = i;
        currentPosition++;
    }
}

void showOutcomes() {
    int shownCount = 0;
    for (int i = 0; i < TOTAL_CONTAINERS; i++) {
        for (int j = 0; j < CONTAINER_SIZE; j++) {
            if (containers[i][j] != SKIP_VALUE) {
                printf("%d ", containers[i][j]);
                shownCount++;

                if (shownCount % SET_SIZE == 0) {
                    printf("\n");
                }
            }
        }
    }
    printf("\n\n"); // little gap at the end
}

int main() {
    // Initialize all containers with SKIP_VALUE
    for (int i = 0; i < TOTAL_CONTAINERS; i++) {
        for (int j = 0; j < CONTAINER_SIZE; j++) {
            containers[i][j] = SKIP_VALUE;
        }
    }

    // Show the initial sequence of symbols
    printf("#### SYMBOLS ####\n");

    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
```

```
    performSteps(number);
    showOutcomes();

    return 0;
}
```

P6: This program computes a desired number of digits of e in any desired base.

```
#include <stdio.h>
#include <stdlib.h>

unsigned char w,h,i,l,e,x,y=10,z=10,E[256];
#define whi1e(j,k)  printf(j,k); fflush(stdout)
#define o E[w]

main (c,v) char *v[]; {
while (c>=2 ){z = atoi(v[1]),--c;
while  (c>=2 )y = atoi(v[2]),--c;}
whi1e ("%s" ,"2.");
while (--y) --x;
while (--x) --e,--y;
while (--z) --x;
while (--x) --e,--z;
while (--w) {
while (--x) --o;}
while (--z) {
while (--x) --w;
while (--o) ;
while (--w) ;
while (--e) --x;
while (--x) --w,--e;
while (--w) {
while (--l) ;
while (--i) --l;--l;
while (--h) ;
while (--y) --x;
while (--x) --h,--y;
while (--x) --h;
while (--h) {
while (--o) --x;
while (--x) --l,--o;
while (l>=w ){--i;
while (--w) --l,--x;
```

```
while (--x) --w;--l;}}
while (--o) ;
while (--l) --x;
while (--x) --o;}
while (--i) --h;
whi1e ("%x" ,--h);--e;}
whi1e ("%s" ,"\n");}
```

Explanation:

The program starts by showing you a short message.
If you've told the program one or two numbers when you began, it keeps these numbers in mind. If you told it only one number, it just remembers that one. If you didn't tell it any numbers, it uses some numbers it already knows.
Then, it does a lot of repetitive tasks, like a person moving beans between bowls or sometimes just throwing beans away, following some specific rules. During these tasks:
1. It occasionally shows you a unique number.
2. At the end, it gives a space.
After all this, the program finishes.

Reconstructed Program for P6:

```
#include <stdio.h>
#define DEFAULT_NUMBER_1 10
#define DEFAULT_NUMBER_2 20
#define REPETITIONS 100

int main() {
    int inputCount, num1, num2;
    printf("Welcome to the program!\n");
    printf("Enter the number of inputs (0, 1, or 2): ");
    scanf("%d", &inputCount);

    switch (inputCount) {
        case 2:
            printf("Enter two numbers: ");
            scanf("%d %d", &num1, &num2);
            break;
        case 1:
            printf("Enter one number: ");
            scanf("%d", &num1);
            num2 = DEFAULT_NUMBER_2;
            break;
        default:
```

```
            num1 = DEFAULT_NUMBER_1;
            num2 = DEFAULT_NUMBER_2;
    }
    for (int i = 1; i <= REPETITIONS; i++) {
        num1 += 2;
        num2 -= 2;
        if (i % 10 == 0) {
            printf("Unique number: %d\n", i);
        }
        if (i % 20 == 0) {
            num1 -= 5;
        }
    }
    printf("\n");
    return 0;
}
```

P7: This program converts byte to binary.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int b,char**i){long long n=B,a=I^n,r=(a/b&a)>>4,y=atoi(*++i)
,_=(((a^n/b)*(y>>T)|y>>S)&r)|(a^r);printf("%.8s\n",(char*)&_);}
```

Explanation:

Imagine you have a machine that takes a number you give it. This machine, using a secret formula and some hidden numbers it already knows, changes your number in various ways. At the end of these changes, the machine sees this result as a word made up of letters and symbols. The machine then shows you only the first part of this word, which is made up of eight characters.

Reconstructed Program for P7:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define MAX_CHARS 8

char predefinedSet[] = "abcdefghijklmnopqrstuvwxyz!@#$%^&*()-_+=";
```

```
uint32_t secretFormula(uint32_t input) {
    // Using some arbitrary hidden numbers and operations
    uint32_t hiddenNum1 = 13;
    uint32_t hiddenNum2 = 7;

    uint32_t result = input * hiddenNum1;
    result += hiddenNum2;
    result ^= (input << 3);
    result -= (hiddenNum1 * hiddenNum2);

    return result;
}

void generateWord(uint32_t result, char *output) {
    size_t setLength = sizeof(predefinedSet) - 1;

    for (int i = 0; i < MAX_CHARS; i++) {
        output[i] = predefinedSet[result % setLength];
        result /= setLength;
    }
    output[MAX_CHARS] = '\0';
}

int main() {
    uint32_t inputNumber;
    char outputWord[MAX_CHARS + 1];

    printf("Enter a number: ");
    scanf("%u", &inputNumber);

    uint32_t result = secretFormula(inputNumber);
    generateWord(result, outputWord);

    printf("The first part of the word is: %s\n", outputWord);

    return 0;
}
```

P8: This program adds two numbers.

```
#      include <stdio.h>
#  define     MAin   printf("%d\n"
```

```
#  define     mAIN return 0
#  define     MaiN  {static
#  define     mAlN   ) {if(
#  define     MA1N    char*
#  define     MAiN     (!!(
#  define     mAiN     atoi
#  define     mAln     &1<<
#  define     MAlN     !=3)
#  define     MAln      )&&
#  define     MAIN      int
#  define     maln      --,
#  define     Maln       <<
#  define     MaIn       ++
#  define     MalN       |=
#  define     MA1n       ||
#  define     malN       -1
#  define     maIN        *
#  define     MaIN        =
#  define     ma1N        )
#  define     Ma1N        (
#  define     Main        ;
#  define     mA1n        !
#  define     MAIn        }
#  define     mA1N        ,
           MAIN     mAIn
  Ma1N     MAIN
  ma1N     mA1N
  mAiN     Ma1N
    MA1N ma1N mA1N maIn MaIN malN mA1N     ma1n
    mA1N   maiN     Main
    MAIN      main Ma1N MAIN Ma1n mA1N MA1N maIN
    mAin      mAlN   Ma1n     MAlN
    mAIN      Main   maIn     MaIn
       mA1N Ma1n maln mAin MaIn   Main     maIn
       MaIN    mAiN   Ma1N     Ma1N
       Ma1n    maln   maIN     mAin
       MaIn    ma1N   ma1N     Main
       ma1n    MaIN   mAiN     Ma1N
       Ma1N    Ma1n   maln     maIN
       mAin    MaIn   ma1N     ma1N
       Main    mAIn   Ma1N     mAIn
       Ma1N    mAIn   Ma1N     mAIn
       Ma1N    mAIn      Ma1N mAIn     Ma1N mAIn
```

```
Ma1N   mAIn      Ma1N mAIn     Ma1N mAIn
  Ma1N mAIn      Ma1N   mAIn  Ma1N  mAIn
    Ma1N mAIn Ma1N      mAIn      Ma1N  mAIn
 Ma1N      Ma1n  ma1N
 ma1N      ma1N  ma1N
  ma1N ma1N ma1N ma1N      ma1N  ma1N
  ma1N  ma1N      ma1N  ma1N
  ma1N  ma1N      Main  MAin
  mA1N maiN ma1N Main mAIN Main      MAIn  MAIN
  mAIn Ma1N   MAIN      mAin  ma1N
  MaiN  MAIN  main        MaIN        malN
  Main   main MaIn  Main      mAIN
  mA1N    maiN        MalN    Ma1N    MAiN
  maIn     mAln      main        ma1N
  MA1n      Ma1N    MAiN        ma1n
  mAln       main  ma1N        MA1n
  mAin  MAln        Ma1N
  mA1n       MAiN  ma1n        mAln
  main       MAln  mAin        ma1N
  ma1N  ma1N    MAln Ma1N mA1n MAiN maIn
  mAln  main    MAln
  Ma1N  MAiN    ma1n
  mAln     main ma1N MA1n mAin MAln
  Ma1N     mA1n MAiN
  ma1n     mAln main
  MAln     mAin ma1N
  ma1N     ma1N ma1N
  ma1N     ma1N Maln
  main     mA1N MAiN
  ma1n     mAln main
  MAln     mAin ma1N
  MA1n     MAiN maIn
  mAln      main        MAln
  Ma1N        MAiN      ma1n
  mAln  main ma1N MA1n
  mAin       MAln
  Ma1N       mA1n
  MAiN       ma1n
  mAln       main
  MAln       mAin
  ma1N       ma1N
  ma1N       ma1N
  Main       MAIn
```

Explanation:

Imagine you have a box of unique toys. This program is like a child playing with these toys, trying out different groups of them. The child takes a few toys out, rearranges them in a specific way, and checks if they fit a special rule. If they do, the child counts how many toys are in that group and shares the number. The child does this for every possible grouping of toys without using the same grouping twice.

Reconstructed Program for P8:

```
#include <stdio.h>
#define TOTAL_TOYS 5
int toys[TOTAL_TOYS] = {1, 2, 3, 4, 5};

int fitsRule(int combination[], int k) {
    int sum = 0;
    for (int i = 0; i < k; i++) {
        sum += combination[i];
    }
    return sum % 2 == 0;
}

void generateCombinations(int start, int k, int combination[]) {
    if (k == 0) {
        if (fitsRule(combination, TOTAL_TOYS - start)) {
            printf("Group of size %d fits the rule!\n",\
            TOTAL_TOYS - start);
        }
        return;
    }

    for (int i = start; i <= TOTAL_TOYS - k; i++) {
        combination[TOTAL_TOYS - k] = toys[i];
        generateCombinations(i + 1, k - 1, combination);
    }
}

int main() {
    for (int i = 1; i <= TOTAL_TOYS; i++) {
        int combination[TOTAL_TOYS];
        generateCombinations(0, i, combination);
    }
```

```
    return 0;
}
```

P9: This program calculates the integer part of square root of the input number.

```
#include <stdio.h>
int l;int f(int o,char **O,int I){char c,*D=O[1];
if(o>0){for(l=0;D[l];D[l++]-=10){D[l++]-=120;D[l]-
=110;while(!f(0,O,l))D[l]+=20;putchar((D[l]+1032)/20);}
putchar(10);}else{c=o+(D[I]+82)%10-(I>l/2)*(D[I-l+I]+72)/10-9;
D[I]+=I<0?0:!(o=f(c/10,O,I-1))*((c+999)%10-(D[I]+92)%10);}
return o;}int main(int argc,char **argv)
{return f(argc,argv,0);}
```

Explanation:

Imagine you have a sequence of special symbols. Each of these symbols has a hidden number attached to it.
1. The program starts by slightly changing the hidden numbers of these symbols based on a secret rule. 2. For every symbol in the sequence, it does the following: - It changes the hidden number of the symbol, using a particular method, until a specific situation occurs. - Once that situation happens, it transforms the hidden number of this symbol into a letter and shows it to you. 3. After revealing the letters for all the symbols in the sequence, it displays one more unique character.
Throughout this process, the program pays attention to the order of the symbols. This is crucial because the middle of the sequence plays a special role. The program uses where a symbol is, compared to the middle, to modify its hidden number in a different way.

Reconstructed Program for P9:

```
#include <stdio.h>
#include <stdlib.h>
#define SEQ_LENGTH 7

int hidden_numbers[SEQ_LENGTH] = {3, 5, 7, 11, 13, 17, 19};

void secret_rule(int index) {
    if (index % 2 == 0)
        hidden_numbers[index]++;
    else
        hidden_numbers[index]--;
```

```
}

void modify_until_specific_situation(int index) {
    int middle = SEQ_LENGTH / 2;
    while (!(hidden_numbers[index] % 2 == 0\
    && hidden_numbers[index] < 10)) {
        if (index < middle)
            hidden_numbers[index] += 2;
        else if (index > middle)
            hidden_numbers[index] -= 2;
        else
            hidden_numbers[index]++;
    }
}

char transform_to_letter(int number) {
    return 'A' + number;
}

int main() {
    for (int i = 0; i < SEQ_LENGTH; i++) {
        secret_rule(i);
    }

    for (int i = 0; i < SEQ_LENGTH; i++) {
        modify_until_specific_situation(i);
        printf("%c ", transform_to_letter(hidden_numbers[i]));
    }

    printf("\nUnique character: #\n");

    return 0;
}
```

P10: A program that plays a two-player tic-tac-toe.

```
#include <stdio.h>

#define N(a)        "%"#a"$hhn"
#define O(a,b)      "%10$"#a"d"N(b)
#define U           "%10$.*37$d"
#define G(a)        "%"#a"$s"
```

```
#define H(a,b)     G(a)G(b)
#define T(a)       a a
#define s(a)       T(a)T(a)
#define A(a)       s(a)T(a)a
#define n(a)       A(a)a
#define D(a)       n(a)A(a)
#define C(a)       D(a)a
#define R          C(C(N(12)G(12)))
#define o(a,b,c)   C(H(a,a))D(G(a))C(H(b,b)G(b))n(G(b))O(32,c)R
#define SS         O(78,55)R "\n\033[2J\n%26$s";
#define E(a,b,c,d) H(a,b)G(c)O(253,11)R G(11)O(255,11)\
R H(11,d)N(d)O(253,35)R
#define S(a,b)     O(254,11)H(a,b)N(68)R G(68)O(255,68)\
N(12)H(12,68)G(67)N(67)

char* fmt = O(10,39)N(40)N(41)N(42)N(43)N(66)N(69)N(24)O(22,65)
O(5,70)O(8,44)N(45)N(46)N    (47)N(48)N(    49)N( 50)N(     51)N(52)
N(53    )O( 28,54)O(5,        55) O(2,    56)O(3,57)O(      4,58 )
O(13,    73)O(4,71 )N(   72)O   (20,59    )N(60)N(61)N 62)N (63)
N    (64)R RE(1,2,   3,13   )E(4,    5,6,13)E(7,8,9        ,13)
E(1,4    ,7,13)E(2,5,8,13)E(    3,6,9,13)E(1,5,         9,13)
E(3    ,5,7,13)E(14,15,    16,23)    E(17,18,19,23)
E(          20, 21,    22,23)E
(14,17,20,23)E(15,    18,21,23)E(16,19,    22     ,23)E(    14, 18,
22,23)E(16,18,20,    23)R U O(255 ,38)R    G (     38)O(    255,36)
R H(13,23)O(255,    11)R H(11,36) O(254    ,36)     R G(    36 ) O(
255,36)R S(1,14    )S(2,15)S(3, 16)S(4,    17 )S     (5,    18)S(6,
19)S(7,20)S(8,    21)S(9    ,22)H(13,23    )H(36,     67    )N(11)R
G(11)""O(255,    25 )R        s(C(G(11)    ))n (G(          11) )G(
11)N(54)R C(    "aa")   s(A(   G(25)))T    (G(25))N         (69)R o
(14,1,26)o(    15, 2,   27)o   (16,3,28    )o( 17,4,        29)o(18
,5,30)o(19    ,6,31)o(        20,7,32)o    (21,8,33)o       (22 ,9,
34)n(C(U)    )N( 68)R H(    36,13)G(23)    N(11)R C(D(      G(11)))
D(G(11))G(68)N(68)R G(68)O(49,35)R H(13,23)G(67)N(11)R C(H(11,11)G(
11))A(G(11))C(H(36,36)G(36))s(G(36))O(32,58)R C(D(G(36)))A(G(36))SS

#define arg d+6,d+8,d+10,d+12,d+14,d+16,d+18,d+20,d+22,\
0,d+46,d+52,d+48,d+24,d+26,d+28,d+30,d+32,d+34,d+36,d+38,d+40,d+50,
(scanf(d+126,d+4),d+(6-2)+18*(1-d[2]%2)+d[4]*2),d,d+66,d+68,d+70,
d+78,d+80,d+82,d+90,d+\
92,d+94,d+97,d+54,d[2],d+2,d+71,d+77,d+83,d+89,d+95,d+72,d+73,d+74\
,d+75,d+76,d+84,d+85,d+86,d+87,d+88,d+100,d+101,d+96,d+102,d+99,d+\
67,d+69,d+79,d+81,d+91,d+93,d+98,d+103,d+58,d+60,d+98,d+126,d+127,\
```

```
d+128,d+129

char d[538] = {1,0,10,0,10};

int main() {
    while(*d) printf(fmt, arg);
}
```

Explanation:

Imagine you have a magical typewriter. This typewriter, instead of typing out stories or letters as you'd expect, types out patterns when given certain clues.
We start with a set of clues. These clues are just a mix of numbers and symbols. The magical typewriter has a special way of reading these clues. It looks at a mix of characters, numbers, and empty spaces, then uses the clues to fill in those empty spaces and decide what to type out. There are different patterns hidden within the clues. Depending on which clues are used, the typewriter might print a star, a dash, or even a sequence of symbols. Sometimes, the typewriter changes the clues a bit based on some rules. It's like changing the recipe slightly to get a slightly different cake. After preparing the clues, the typewriter types out patterns, one after another, creating a series of designs on the paper. So, the end result is a paper filled with unique patterns, all determined by the initial clues and the magical typewriter's special way of reading them.

Reconstructed Program for P10:

```
#include <stdio.h>
#include <string.h>

void encrypt(char clues[]) {
    int len = strlen(clues);
    char result[len + 1]; // +1 for the null-terminator
    result[len] = '\0';
    for (int i = 0; i < len; i++) {
        int value = (int)clues[i];

        if (value % 3 == 0) {
            result[i] = '*';
        } else if (value % 5 == 0) {
            result[i] = '-';
        } else {
            result[i] = '+';
```

```
        }
    }

    for (int i = 0; i < len; i++) {
        int value = (int)clues[i];
        if (value % 2 == 0) {
            for (int j = 0, k = i; j < k; j++, k--) {
                char temp = result[j];
                result[j] = result[k];
                result[k] = temp;
            }
        }
    }
    printf("Encrypted Pattern: %s\n", result);
}

int main() {
    char clues[100];
    printf("Enter your clues (numbers & symbols): ");
    scanf("%s", clues);
    encrypt(clues);
    return 0;
}
```

B.4.2 Non-obfuscated Equivalent Programs

P1: This program computes all proper subsets of the set of arguments passed to it.

```
#include <stdio.h>

void generate_subsets(char *argv[], int current, int n, \
char *subset[], int subsetSize) {
    if (current == n) {
        if (subsetSize < n) {
            for (int i = 0; i < subsetSize; i++) {
                printf("%s ", subset[i]);
            }
            printf("\n");
        }
        return;
    }
```

```
    generate_subsets(argv, current + 1, n, subset, subsetSize);
    subset[subsetSize] = argv[current];
    generate_subsets(argv, current + 1, n, subset, subsetSize + 1);
}

int main(int argc, char *argv[]) {
    if (argc <= 1) {
        printf("Please provide some elements to generate subsets.");
        return 1;
    }
    char *subset[argc-1]; // This will store the current subset
    printf("Proper subsets:\n");
    generate_subsets(argv+1, 0, argc-1, subset, 0);

    return 0;
}
```

Explanation:

Imagine you have a magical box. You give this box several items, one after the other. Once done, the box has the power to reveal all possible groupings of the items you've given it, without ever showing every single item together at once. For instance, if you give the box an apple, a banana, and a cherry, the box can reveal groupings like:
Only the apple. Only the banana. Only the cherry. The apple with the banana. The apple with the cherry. The banana with the cherry. But it will never show you the apple, banana, and cherry all together since that's exactly what you gave it to start with.
If you approach the magical box and don't give it any items, it will kindly tell you to provide some items so it can show the groupings.

Reconstructed Program for non-obfuscated P1:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_ITEMS 10  // Assume a maximum of 10 items for simplicity

int itemCount;
char *items[MAX_ITEMS];

void displayGroupings(int idx, bool included[MAX_ITEMS],
int includedCount) {
    if (idx == itemCount) {
        if (includedCount > 0 && includedCount < itemCount) {
```

```
            for (int i = 0; i < itemCount; i++) {
                if (included[i]) {
                    printf("%s ", items[i]);
                }
            }
            printf("\n");
        }
        return;
    }

    included[idx] = true;
    displayGroupings(idx + 1, included, includedCount + 1);

    included[idx] = false;
    displayGroupings(idx + 1, included, includedCount);
}

int main() {
    printf("Enter the number of items (max 10): ");
    scanf("%d", &itemCount);

    if (itemCount < 1 || itemCount > MAX_ITEMS) {
        printf("Invalid number of items.\n");
        return 1;
    }

    // Input each item name
    printf("Provide the items:\n");
    for (int i = 0; i < itemCount; i++) {
        char *item = (char *)malloc(50 * sizeof(char));
        scanf("%s", item);
        items[i] = item;
    }

    bool included[MAX_ITEMS] = {false};
    displayGroupings(0, included, 0);

    // Free the allocated memory
    for (int i = 0; i < itemCount; i++) {
        free(items[i]);
    }

    return 0;
```

```
}
```

P2: This program calculates the value of π.

```
#include <stdio.h>
#include <stdlib.h>

long insideCircleCount = 0, totalPoints = 0;

void computePi() {
    for (int i = 0; i < 10000; i++) {
        double x = (double) rand() / RAND_MAX;
        double y = (double) rand() / RAND_MAX;

        if (x * x + y * y <= 1) {
            insideCircleCount++;
        }

        totalPoints++;
    }
}

int main() {
    computePi();
    printf("%1.3f\n", 4. * insideCircleCount / totalPoints);
    return 0;
}
```

Explanation:

Imagine you have a square dartboard with a circle perfectly drawn inside it. The circle touches all four sides of the square. Now, you decide to throw darts randomly onto this board 10,000 times. Each time you throw a dart, you make a note of whether the dart landed inside the circle or outside it but still inside the square. After all your throws, you count how many darts landed inside the circle.
Using this information, you can then figure out a special number by taking the count of darts that landed inside the circle, multiplying it by 4, and then dividing it by the total number of darts you threw. The machine then displays this special number up to three decimal places.

Reconstructed Program for non-obfuscated P2:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

#define TOTAL_DARTS 10000

bool isInsideCircle(double x, double y) {
    return (x * x + y * y) <= 1.0;
}

int main() {
    int countInsideCircle = 0;
    double x, y;

    // Seed the random number generator
    srand(time(NULL));

    for (int i = 0; i < TOTAL_DARTS; i++) {
        x = (double)rand() / RAND_MAX * 2.0 - 1.0;
        y = (double)rand() / RAND_MAX * 2.0 - 1.0;

        if (isInsideCircle(x, y)) {
            countInsideCircle++;
        }
    }

    double estimatedPi = 4.0 * countInsideCircle / TOTAL_DARTS;
    printf("Estimated value of pi: %.3f\n", estimatedPi);

    return 0;
}
```

P3: This program prints the input string sorted alphabetically.

```
#include <stdio.h>
#include <string.h>

void sortStringAlphabetically(char str[]) {
    int len = strlen(str);
```

```
    char temp;
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (str[j] > str[j + 1]) {
                // Swap characters if they are out of order
                temp = str[j];
                str[j] = str[j + 1];
                str[j + 1] = temp;
            }
        }
    }
}

int main() {
    char inputString[100];

    printf("Enter a string: ");
    fgets(inputString, sizeof(inputString), stdin);

    // Remove the newline character at the end of the string
    if (inputString[strlen(inputString) - 1] == '\n') {
        inputString[strlen(inputString) - 1] = '\0';
    }

    sortStringAlphabetically(inputString);

    printf("Sorted string: %s\n", inputString);

    return 0;
}
```

Explanation:

Imagine you have a collection of letters, and you'd like to rearrange these letters in a specific order similar to how you would find them in a dictionary. First, you're asked to provide this collection of letters. After entering them, the program will then work on rearranging them. It will repeatedly compare two adjacent letters and swap them if they are not in the desired order. This process of comparison and swapping is repeated multiple times until all the letters are in their proper positions.
Once the letters are rearranged, the program will show you the new arrangement.

Reconstructed Program for non-obfuscated P3:

```
#include <stdio.h>
#include <string.h>

void bubbleSort(char arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // swap arr[j] and arr[j+1]
                char temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    char input[100];

    printf("Enter the collection of letters: ");
    scanf("%s", input);
    int length = strlen(input);
    bubbleSort(input, length);
    printf("Sorted letters in dictionary order: %s\n", input);

    return 0;
}
```

P4: This program prints the first 100 prime numbers.

```
#include <stdio.h>
#include <stdbool.h>

bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
```

```
    }
    return true;
}

int main() {
    int count = 0;  // Count of prime numbers found
    int number = 2;  // Starting number for the search

    printf("The first 100 prime numbers are:\n");

    while (count < 100) {
        if (isPrime(number)) {
            printf("%d ", number);
            count++;
        }
        number++;
    }

    printf("\n");
    return 0;
}
```

Explanation:

Imagine you have a machine that is searching for very special numbers. These numbers are unique because they cannot be divided evenly by any number other than 1 and themselves.
When you start the machine, it begins its search with the first whole number after 1. It checks to see if this number is special. If it is, the machine displays it. The machine then continues to the next number and does the same check. This machine is persistent and continues its search, showing you each special number it finds. It will stop after it has shown you a hundred of these special numbers.

Reconstructed Program for non-obfuscated P4:

```
#include <stdio.h>
#include <stdbool.h>

bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
```

```
    return true;
}

int main() {
    int count = 0;
    int number = 2;
    while (count < 100) {
        if (isPrime(number)) {
            printf("%d ", number);
            count++;
        }
        number++;
    }
    printf("\n");

    return 0;
}
```

P5: This program computes the factorial of a number.

```
#include<stdio.h>
int main()
{
 int i,fact=1,number;
 printf("Enter a number: ");
  scanf("%d",&number);
    for(i=1;i<=number;i++){
      fact=fact*i;
  }
  printf("Factorial of %d is: %d",number,fact);
return 0;
}
```

Explanation:

Imagine a machine that helps you find the product of a series of counting numbers up to a given number. When you approach this machine, it asks you to tell it a specific number. Once you provide the number, the machine starts multiplying all the counting numbers from 1 up to the number you provided. For example, if you tell the machine the number 4, it will multiply 1 by 2, then multiply the result by 3, and then multiply that result by 4.
After finishing the calculations, the machine displays a message that tells you

the result of multiplying all those counting numbers up to the number you provided.

Reconstructed Program for non-obfuscated P5:

```
#include <stdio.h>
unsigned long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    if (number < 0) {
        printf("Please enter a non-negative number.\n");
        return 1;
    }
    unsigned long long result = factorial(number);
    printf("Result", result);

    return 0;
}
```

P6: This program computes a desired number of digits of e in any desired base.

```
#include <stdio.h>

#define MAX_DIGITS 1000

int result[MAX_DIGITS];
int temp[MAX_DIGITS];
int factorial[MAX_DIGITS];

void add(int *a, int *b) {
    for (int i = 0; i < MAX_DIGITS; i++) {
        a[i] += b[i];
```

```
        if (a[i] >= 10) {
            a[i+1] += a[i] / 10;
            a[i] %= 10;
        }
    }
}

void computeE(int digits) {
    result[0] = 1;
    factorial[0] = 1;

    for (int i = 1; i <= digits; i++) {
        for (int j = 0; j < MAX_DIGITS; j++) {
            temp[j] = factorial[j] = factorial[j] * i;
        }

        for (int j = 0; j < MAX_DIGITS; j++) {
            if (temp[j] >= 10) {
                temp[j+1] += temp[j] / 10;
                temp[j] %= 10;
            }
        }
        add(result, temp);
    }
}

void convertAndPrint(int base) {
    int output[MAX_DIGITS] = {0};
    int idx = 0;

    while (idx < MAX_DIGITS) {
        int remainder = 0;
        for (int i = MAX_DIGITS - 1; i >= 0; i--) {
            int temp = result[i] + remainder * 10;
            result[i] = temp / base;
            remainder = temp % base;
        }
        output[idx++] = remainder;
        int sum = 0;
        for (int i = 0; i < MAX_DIGITS; i++) {
            sum += result[i];
        }
        if (sum == 0) break;
```

```
    }

    for (int i = idx - 1; i >= 0; i--) {
        printf("%d ", output[i]);
    }
    printf("\n");
}

int main() {
    int desired_digits, base;
    printf("Enter the desired number of digits: ");
    scanf("%d", &desired_digits);
    printf("Enter the desired base (2 to 10): ");
    scanf("%d", &base);

    if (base < 2 || base > 10) {
        printf("Base out of range.\n");
        return 1;
    }

    computeE(desired_digits);
    convertAndPrint(base);
    return 0;
}
```

Explanation:

Imagine a machine that can tell you about a special number, much like pi, which starts 2.71828... and goes on forever. The unique thing about this machine is that it can show this special number using different number systems, not just the one we usually use with numbers 0 to 9.
When you approach the machine, it asks you two things: 1. How many parts of this special number do you want to know? You tell it a number. 2. In which number system you'd like to see this special number? For this, you can choose any system between the one that uses just 0 and 1 and the usual one (that uses numbers 0 to 9).
Once you provide this information, the machine works its magic. It calculates the value of this special number up to the number of parts you specified, in the number system you chose. Then, it displays that special value for you.
However, if you choose a number system outside the allowed range, the machine will tell you that the choice is not acceptable and stop.

Reconstructed Program for non-obfuscated P6:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <math.h>

double calculateE(int parts) {
    double e = 1.0;
    double factorial = 1.0;
    for (int i = 1; i < parts; i++) {
        factorial *= i;
        e += 1.0 / factorial;
    }
    return e;
}

void displayInBase(double number, int base, int parts) {
    printf("2."); // Display the integer part
    for (int i = 0; i < parts; i++) {
        number *= base;
        int digit = (int) number;
        printf("%d", digit);
        number -= digit;
    }
    printf("\n");
}

int main() {
    int parts, base;
    printf("How many parts of e do you want to know? ");
    scanf("%d", &parts);
    printf("In which base system (between 2 and 9 inclusive)? ");
    scanf("%d", &base);
    if (base < 2 || base > 9) {
        printf("The chosen number system is not acceptable.\n");
        return 1;
    }
    double e = calculateE(parts + 5);
    displayInBase(e - 2.0, base, parts);

    return 0;
}
```

P7: This program converts byte to binary.

```
#include <stdio.h>
void byteToBinary(unsigned char byte) {
    for (int i = 7; i >= 0; i--) {
        printf("%d", (byte >> i) & 1);
    }
    printf("\n");
}

int main() {
    unsigned char inputByte;
    printf("Enter a byte value (0-255): ");
    scanf("%hhu", &inputByte);

    printf("Binary representation: ");
    byteToBinary(inputByte);
    return 0;
}
```

Explanation:

Imagine you have a line of 8 light bulbs, all turned off. These light bulbs can either be on (showing a light) or off (no light). Now, imagine you have a number between 0 and 255. Based on this number, some of these light bulbs will turn on, while others will remain off. This program is like a guidebook. First, it asks you for that number. Once you give the number, the guidebook will show you a line indicating which light bulbs should be turned on and which should remain off, using a sequence of 1s (for the bulbs that are on) and 0s (for the bulbs that are off). By looking at this sequence, you will be able to recreate the exact arrangement of lit and unlit light bulbs for the number you provided.

Reconstructed Program for non-obfuscated P7:

```
#include <stdio.h>
int main() {
    int number;
    printf("Enter a number between 0 and 255: ");
    scanf("%d", &number);
    if (number < 0 || number > 255) {
        printf("Invalid number.\
        Please enter a number between 0 and 255.\n");
        return 1;
    }
```

```
    printf("Light bulb sequence: ");
    for (int i = 7; i >= 0; i--) {
        if ((number & (1 << i)) != 0) {
            printf("1");
        } else {
            printf("0");
        }
    }
    printf("\n");
    return 0;
}
```

P8: This program adds two numbers.

```
#include <stdio.h>

int main() {
    int num1, num2, sum;

    printf("Enter the first number: ");
    scanf("%d", &num1);

    printf("Enter the second number: ");
    scanf("%d", &num2);

    sum = num1 + num2;

    printf("The sum of %d and %d is %d.\n", num1, num2, sum);

    return 0;
}
```

Explanation:

Imagine you have a digital assistant with a screen. When you turn it on, it first asks you to type in a number. After you've provided the first number, it asks you for a second one. Once you've entered both numbers, the assistant does a quick mental calculation and shows you the result of adding these two numbers together. It displays a message like "The sum of the first number and the second number is the result." With this message, you immediately know how much the two numbers you provided add up to.

Reconstructed Program for non-obfuscated P8:

```
#include <stdio.h>

int main() {
    int firstNumber, secondNumber, sum;
    printf("Enter the first number: ");
    scanf("%d", &firstNumber);
    printf("Enter the second number: ");
    scanf("%d", &secondNumber);
    sum = firstNumber + secondNumber;
    printf("The sum of the first number (%d) and \
    the second number (%d)is %d.\n",\
    firstNumber, secondNumber, sum);

    return 0;
}
```

P9: This program calculates the integer part of square root of the input number.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int main() {
    char number[100];  // buffer for input number
    printf("Enter a number with even number of digits: ");
    scanf("%s", number);

    int len = strlen(number);
    if (len % 2 != 0) {
        printf("The number must have an even number of digits.\n");
        return 1;
    }

    double value = atof(number);
    double sqrtValue = sqrt(value);

    printf("Square root: %d\n", (int)sqrtValue);
    return 0;
```

```
}
```

Explanation:

Imagine a digital tool that helps you find the square root of numbers. When you start it, it asks you to type in a special kind of number: one that has an even count of digits. For example, if you have a number like "1234", it's okay because it has 4 digits. But a number like "123" won't work since it has 3 digits. After you type in this number, the tool checks to make sure it has the right amount of digits. If the number you typed doesn't follow the rule, the tool will tell you that you've made a mistake and need to type in a number with an even count of digits.
However, if the number you typed is correct, the tool will quickly do some calculations and then show you the square root of that number. But, it only shows you the whole part of the square root, without any decimals.

Reconstructed Program for non-obfuscated P9:

```
#include <stdio.h>
#include <math.h>
#include <string.h>

int main() {
    char numberString[100];
    int isValid = 1;
    printf("Enter a number with an even count of digits: ");
    scanf("%s", numberString);
    int length = strlen(numberString);
    if (length % 2 != 0) {
        printf("The number you entered has an odd count of digits.\
        Please enter a number with an even count of digits.\n");
        isValid = 0;
    }

    if (isValid) {
        double num = atof(numberString);
        int squareRootWholePart = (int)sqrt(num);
        printf("The whole part of the square root\
        of %s is %d.\n", numberString, squareRootWholePart);
    }

    return 0;
}
```

P10: A program that plays a two-player tic-tac-toe.

```
#include <stdio.h>
#define SIZE 3

// Function prototypes
void displayBoard(char board[SIZE][SIZE]);
int checkWinner(char board[SIZE][SIZE]);
void getPlayerMove(char board[SIZE][SIZE], char player);

int main() {
    char board[SIZE][SIZE] = {{' ', ' ', ' '},\
    {' ', ' ', ' '}, {' ', ' ', ' '}};
    int moves = 0;
    char currentPlayer = 'X';

    displayBoard(board);

    while (moves < SIZE * SIZE) {
        getPlayerMove(board, currentPlayer);

        displayBoard(board);

        if (checkWinner(board)) {
            printf("Player %c wins!\n", currentPlayer);
            return 0;
        }

        currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
        moves++;
    }

    printf("It's a draw!\n");
    return 0;
}

void displayBoard(char board[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%c", board[i][j]);
            if (j < SIZE - 1) printf(" | ");
        }
        printf("\n");
```

```
        if (i < SIZE - 1) printf("---------\n");
    }
}

void getPlayerMove(char board[SIZE][SIZE], char player) {
    int x, y;
    do {
        printf("Player %c, enter your move \
        (row [0-2] and column [0-2]): ", player);
        scanf("%d%d", &x, &y);
    } while (x < 0 || x >= SIZE || \
    y < 0 || y >= SIZE || board[x][y] != ' ');

    board[x][y] = player;
}

int checkWinner(char board[SIZE][SIZE]) {
    // Check rows, columns and diagonals
    for (int i = 0; i < SIZE; i++) {
        if (board[i][0] != ' ' && board[i][0] == board[i][1] \
        && board[i][1] == board[i][2]) return 1;
        if (board[0][i] != ' ' && board[0][i] == board[1][i] \
        && board[1][i] == board[2][i]) return 1;
    }
    if (board[0][0] != ' ' && board[0][0] == board[1][1] \
    && board[1][1] == board[2][2]) return 1;
    if (board[0][2] != ' ' && board[0][2] == board[1][1] \
    && board[1][1] == board[2][0]) return 1;

    return 0;
}
```

Explanation:

Imagine a classic game where two players take turns marking spaces on a 3x3 grid. One player uses an 'X' and the other uses an 'O'. The goal of the game is to mark three spaces in a row, either horizontally, vertically, or diagonally before the opponent does.
When the game starts, the grid is empty. The game tool displays the grid to show both players the current state of the game. Each space in the grid can be identified using two numbers: the row number and the column number, both ranging from 0 to 2.
One player goes first, marking a space on the grid with their symbol. The

game tool then updates the grid to show this mark. After each turn, the tool checks if there are any three matching symbols in a row. If there is a winning line, it announces which player has won.
If all the spaces on the grid are marked, and no player has three of their symbols in a row, then the game is declared a draw.
The game continues by allowing players to take turns marking spaces until someone wins or all spaces are marked, resulting in a draw.

Reconstructed Program for non-obfuscated P10:

```
#include <stdio.h>
#include <stdbool.h>

char board[3][3] = {{' ', ' ', ' '},
                    {' ', ' ', ' '},
                    {' ', ' ', ' '}};

bool checkWin(char player) {
    for (int i = 0; i < 3; i++) {
        if (board[i][0] == player && board[i][1] == player &&\
        board[i][2] == player) return true;
        if (board[0][i] == player && board[1][i] == player &&\
        board[2][i] == player) return true;
    }
    if (board[0][0] == player && board[1][1] == player &&\
    board[2][2] == player) return true;
    if (board[0][2] == player && board[1][1] == player &&\
    board[2][0] == player) return true;

    return false;
}

bool checkDraw() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == ' ') return false;
        }
    }
    return true;
}

void displayBoard() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

```
            printf("%c", board[i][j]);
            if (j < 2) printf(" | ");
        }
        printf("\n");
        if (i < 2) printf("---------\n");
    }
}

int main() {
    char currentPlayer = 'X';
    int row, col;

    while (true) {
        displayBoard();

        printf("Player %c, enter row (0-2)\
        and column (0-2): ", currentPlayer);
        scanf("%d %d", &row, &col);

        if (row >= 0 && row < 3 && col >= 0 &&\
        col < 3 && board[row][col] == ' ') {
            board[row][col] = currentPlayer;

            if (checkWin(currentPlayer)) {
                displayBoard();
                printf("Player %c wins!\n", currentPlayer);
                break;
            }

            if (checkDraw()) {
                displayBoard();
                printf("It's a draw!\n");
                break;
            }

            currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
        } else {
            printf("Invalid move, try again.\n");
        }
    }

    return 0;
}
```

B.5 Useless code snippets added for Obfuscation

B.5.1 Synthesized Set

B.5.1.1 Level 1

```
if (u.canHarvest()) then {
    for (unit u){
        if (u.isBuilder()) then{
    }
    else{
        u.harvest(50);
            }
        }
}
for (Unit u){
    if (u.is_Type(Worker)) then{
        u.train(Heavy, Enemydir,10);
        if (u.canAttack()) then {
            u.train(Ranged,Up,16)
        }
    }
}
```

B.5.1.2 Level 2

```
if (u.canHarvest()) then {
    for (unit u){
        u.train(Heavy,Left, 9)
        if (u.isBuilder()) then{

        }
        else{
            u.harvest(50);
        }
    }
}
for (Unit u){
    if (u.is_Type(Worker)) then{
```

```
        u.train(Heavy, Enemydir,10);
        if (u.canAttack()) then {
            u.train(Ranged,Up,16)
            if (u.canHarvest()) then {
                u.train(Worker,Right, 9)
                if (u.isBuilder()) then{
                    if (u.is_Type(Barracks)) then{
                        u.harvest(10);
                        u.train(Workers, Right, 2);
                    }
                    else{
                        u.train(Light,Down,5)
                        }
                }

            }
        }
    }
    else{
        u.harvest(1);
    }
    for(Unit u){
        u.idle()
    }
}
for (Unit u){
    if(u.hasLessNumberOfUnits(Heavy, 2)){
        if (u.isBuilder()) then{
            u.train(Light, Farthest, 1);
        }
        else{
            u.harvest(3);
        }

    }
    if (hasNumberOfUnits(Base, 3)){
        if (u.is_Type(Barracks)) then{
            u.build(Base,EnemyDir,1);
            u.moveToUnit(Ally, MostHealthy);
        }
        else{
            u.train(Heavy, Random,10);
        }
```

```
    }
}
```

B.5.2 Human-crafted Set

B.5.2.1 Level 1

```
if (u.getType()==barracksType && u.getType().canHarvest &&\
p.getResources() <= heavyType.cost){
    train(u, heavyType);
    buildIfNotAlreadyBuilding(u,baseType,u.getX(),u.getY(),\
    reservedPositions,p,pgs);
    }
int nlight = 0;
for (Unit u4 : pgs.getUnits()) {
    if (u4.getType() == lightType && u4.getPlayer() != p.getID()) {
        nlight++;
    }
}
if (nlight < 2 && u.getType().canMove) {
    if (u.getType().canHarvest){
        train(u, lightType);
    }
}
Unit closestBase = null;
Unit closestResource = null;
int closestDistance = 0;
for (Unit u3 : pgs.getUnits()) {
    if (u3.getType().isResource) {
        int d = Math.abs(u3.getX() - u.getX()) +\
        Math.abs(u3.getY() - u.getY());
        if (closestResource == null || d < closestDistance) {
            closestResource = u3;
            closestDistance = d;
        }
    }
}
closestDistance = 0;
for (Unit u4 : pgs.getUnits()) {
    if (u4.getType().isStockpile && u4.getPlayer()==p.getID()) {
        int d = Math.abs(u4.getX() - u.getX()) +\
        Math.abs(u4.getY() - u.getY());
        if (closestBase == null || d < closestDistance) {
```

```
            closestBase = u4;
            closestDistance = d;
        }
    }
}
if (closestResource != null && closestBase != null &&\
u.getType().canHarvest) {
    harvest(u, closestResource, closestBase);
}
if (u.getType().canMove == true &&\
p.getResources() >= workerType.cost) {
    buildIfNotAlreadyBuilding(u,baseType,u.getX(),u.getY(),\
    reservedPositions,p,pgs);
    train(u, heavyType);
}
```

B.5.2.2 Level 2

```
Unit TargetEnemy = null;
int Distance = 0;
int Mybase = 0;
int Mybarrack = 0;
for (Unit u6 : pgs.getUnits()) {
    if (u6.getPlayer() >= 0 && u6.getPlayer() != p.getID()) {
        int d = Math.abs(u6.getX() - u.getX()) +\
        Math.abs(u6.getY() - u.getY());
        if (TargetEnemy == null || d > Distance) {
            TargetEnemy = u6;
            Distance = d;
        }
    }
    else if(u6.getPlayer()==p.getID() &&\
    u6.getType() == baseType){
        Mybase = Math.abs(u6.getX() - u.getX()) +\
        Math.abs(u6.getY() - u.getY());
    }
    else if(u6.getPlayer()==p.getID() &&\
    u6.getType() == barracksType){
        Mybarrack = Math.abs(u6.getX() - u.getX()) +\
        Math.abs(u6.getY() - u.getY());
    }
}
if (u.getType() == workerType && TargetEnemy!=null &&\
```

```
(Distance < pgs.getHeight()/2 || Mybase < pgs.getHeight()/2)) {
    attack(u, TargetEnemy);
}
else if(u.getType() == workerType &&\
!(Distance < pgs.getHeight()/2 || Mybase < pgs.getHeight()/2)){
    buildIfNotAlreadyBuilding(u, barracksType, u.getX(),u.getY(),\
    reservedPositions,p,pgs);
}
else if(u.getType() == workerType &&\
!(Distance < pgs.getHeight()/2 || Mybase < pgs.getHeight()/2)){
    buildIfNotAlreadyBuilding(u, baseType, u.getX(),u.getY(),\
    reservedPositions,p,pgs);
}
for (Unit u5 : pgs.getUnits()) {
    if (u5.getType() == baseType || u5.getType() == barracksType ){
        if (u5.getPlayer() == p.getID()){
            if (TargetEnemy!=null && u.getType() == workerType &&(\
            Distance>pgs.getHeight()/2 || Mybase>pgs.getHeight()/2))
            {
                attack(u5, TargetEnemy);
                if (u5.getType()==workerType){
                    train(u, lightType);
                }
            }
        }
    }
}
if (u.getType()==barracksType && u.getType().canHarvest &&\
p.getResources() <= heavyType.cost){
    train(u, heavyType);
    buildIfNotAlreadyBuilding(u,baseType,u.getX(),u.getY(),\
    reservedPositions,p,pgs);
}
int nlight = 0;
for (Unit u4 : pgs.getUnits()) {
    if (u4.getType() == lightType && u4.getPlayer() != p.getID()) {
        nlight++;
    }
}
if (nlight < 2 && u.getType().canMove) {
    if (u.getType().canHarvest){
        train(u, lightType);
    }
```

```
}
Unit closestBase = null;
Unit closestResource = null;
int closestDistance = 0;
for (Unit u3 : pgs.getUnits()) {
    if (u3.getType().isResource) {
        int d = Math.abs(u3.getX() - u.getX()) +\
        Math.abs(u3.getY() - u.getY());
        if (closestResource == null || d < closestDistance) {
            closestResource = u3;
            closestDistance = d;
        }
    }
}
closestDistance = 0;
for (Unit u4 : pgs.getUnits()) {
    if (u4.getType().isStockpile && u4.getPlayer()==p.getID()) {
        int d = Math.abs(u4.getX() - u.getX()) +\
        Math.abs(u4.getY() - u.getY());
        if (closestBase == null || d < closestDistance) {
            closestBase = u4;
            closestDistance = d;
        }
    }
}
if (closestResource != null && closestBase != null &&\
u.getType().canHarvest) {
    harvest(u, closestResource, closestBase);
}
if (u.getType().canMove == true &&\
p.getResources() >= workerType.cost) {
    buildIfNotAlreadyBuilding(u,baseType,u.getX(),u.getY(),\
    reservedPositions,p,pgs);
    train(u, heavyType);
}
```

B.5.2.3 Justification

The main reason we need more lines to break down programs in the human-crafted set compared to the synthesized set, is due to the language they are written in. The human-crafted set is written in Java, a language that the LLM knows very well. This makes it difficult to confuse or mislead the LLM with

Java. We observed that GPT-4 is particularly good at reconstructing Java code from explanations.The process of explaining is also more straightforward with Java, as GPT-4 finds it easier to understand and translate Java into explanations compared to a newly defined language. Consequently, we required a greater number of lines to sufficiently obfuscate the programs.

B.6 Set of MicroRTS Programs

B.6.1 Synthesized Set

P1:

```
for(Unit u){
    for(Unit u){
        u.harvest(2)
        u.idle()
    }
    u.train(Light,Up,25)
    u.train(Worker,Right,4)
    u.attack(Closest)
    for(Unit u){
        u.train(Heavy,Up,3)
    }
    for(Unit u){
        u.build(Barracks,Up,1)
    }
}
```

Explanation:

To get started, the very first action this strategy takes with topmost priority involves sending each of our units to gather resources. Specifically, every individual unit is assigned to harvest a maximum of two resources. After that, these units are instructed to remain idle. The idle instruction is crucial because it ensures that while these units are not engaged in harvesting, they still remain vigilant, ready to attack any opponent unit that comes within their attack range.
After the initial phase of resource harvesting, the next priority of this strategy is to train new units. Initially, we aim to train up to 25 Light units in the upward direction from their respective training structures. Following this, we have a target to train up to 4 Worker units who would be placed on the right side of their training structure.

The next action, which comes after the training of Light and Worker units, is attacking the opponent units. Here, the strategy is to attack the closest opponent unit. Hence, if there are several opponent units within the attack range, the closest one will be targeted first.
Subsequently, we have two priorities that occur concurrently. On one hand, we aim to train a limited number of Heavy units (up to 3 in number). On the other hand, we look to construct Barracks with a maximum limit of 1 Barracks. The placement for both these actions is towards the upward direction from the current position.
In summary, the overall goal of this strategy is to maintain an equilibrium between resource harvesting, unit training, and combating the enemy. By prioritizing resource gathering initially, we ensure that there are enough resources available for training new units and building structures. Following that, we put a focus on training both combatant (Light and Heavy) and resource gathering (Worker) units, thereby maintaining the balance in our forces. Finally, the strategy underscores the importance of continuous attacks on the closest enemy while simultaneously enhancing our forces through the construction of Barracks and training of Heavy units.

Best Reconstructed Program for P1:

```
for(Unit u) {
    u.harvest(10);
    u.idle();
}
for(Unit u) {
    u.train(Light, Up, 25);
}
for(Unit u) {
    u.train(Worker, Right, 4);
    u.attack(Closest);
}
for(Unit u) {
    u.train(Heavy, Up, 3);
}
for(Unit u) {
    u.build(Barracks, Up, 1);
}
```

P2:

```
for(Unit u){
u.build(Barracks,EnemyDir,7)
```

```
u.harvest(9)
for(Unit u){
u.train(Heavy,Left,20)
}
u.train(Worker,Right,3)
u.attack(Weakest)
for(Unit u){
u.idle()
}
u.moveToUnit(Ally,LessHealthy)
for(Unit u){
u.train(Light,Left,2)
}
}
```

Explanation:

The provided strategy for playing MicroRTS takes a structured approach, prioritizing certain actions over others in an attempt to gain an advantage. It is important to understand the priority of actions to follow and execute this strategy properly. The initial and highest priority of this strategy is to establish a strong defense by building Barracks towards the enemy's direction. The aim here is to construct 7 Barracks, which would serve as the primary base for training combat units. At the same time, the strategy also assigns priority to resource collection, instructing workers to harvest resources, targeting 9 resource units.

The next action with high priority is to train Heavy units from the units within the Barracks. These units are positioned on the left of the Barracks. The training aims to produce up to 20 Heavy units which would serve as the main force in our offensive and defensive operations.

Subsequently, the strategy focuses on replenishing and expanding the workforce. The Barracks are instructed to train more Workers on the right side, targeting a total of 3 additional Workers. These additional workers would serve to increase the efficiency of resource collection and building processes.

With the fortifications and army in place, the strategy now shifts towards offense. The units are directed to attack the weakest of the opponent's units. This action is aimed at diminishing the opponent's strength and defenses by targeting their vulnerable units first.

After the attack command, the next action with priority is to direct units to remain idle. This action ensures that the units conserve their energy and are prepared for any sudden attacks from the opponent.

The strategy then focuses on ensuring the survival of ally units. Units are instructed to move towards the least healthy ally unit. This action might

help in providing reinforcement or drawing away enemy units from the weaker allies.
The final action in this strategy is to train Light units. Light units are quicker to train and would serve as a rapid response force. The strategy aims to train 2 Light units on the left side of the Barracks.
In summary, the overall goal of this strategy is to build a strong defense by quickly constructing Barracks and training Heavy units, maintain a steady flow of resources, selectively target vulnerable enemy units while ensuring the survival of ally units, and lastly, training a rapid response force of Light units.

Best Reconstructed Program for P2:

```
for(Unit u){
    for(Unit u){
        u.build(Barracks, EnemyDir, 7)
        u.harvest(9)
        u.train(Heavy, Left, 20)
    }
    for(Unit u){
        u.train(Worker, Right, 3)
        u.attack(Weakest)
        u.idle()
        u.moveToUnit(Ally, LessHealthy)
        u.train(Light, Left, 2)
    }
}
```

P3:

```
for(Unit u){
    for(Unit u){
        u.idle()
    }
    u.moveToUnit(Enemy,Closest)
    u.train(Worker,EnemyDir,2)
    for(Unit u){
        u.train(Heavy,EnemyDir,100)
        u.train(Light,Left,8)
        u.attack(Weakest)
        if(u.OpponentHasUnitInPlayerRange()) then {
            u.moveToUnit(Enemy,LessHealthy)
        }
```

```
        for(Unit u){
            u.build(Barracks,EnemyDir,20)
            u.harvest(6)
            u.attack(LessHealthy)
            u.moveToUnit(Ally,Weakest)
        }
    }
}
```

Explanation:

The strategic gameplay outlined in program P prioritizes actions to maximize efficiency and tactical advantage in the MicroRTS game. The program begins with the top priority action, instructing every unit in the player's control to stay idle. This implies that unless a unit is immediately threatened, it should initially hold its position and avoid proactive movement or combat. After this first instruction, the player's units are instructed to relocate towards the nearest enemy unit, in order to prepare for an impending confrontation. This directive stands as the second priority in the strategy.
The third priority is to utilize the player's bases to train two worker units in the direction of the enemy. This is intended to increase the player's resource collection capability while also adding a mild defensive presence against incoming enemy units.
Next, for every unit, it prioritizes training a large army of heavy combat units directed towards the enemy. If this is not feasible (e.g., the unit in question is not a barracks), it then looks to train light combat units to the left. After training these units, the next action prioritized is for them to target and attack the weakest enemy units.
Within this framework, if the player's unit detects an enemy unit within its attack range, the unit will change its course and move towards the less healthy enemy unit. This dynamic movement decision is prioritized after the initial set of actions.
Further, for every unit, there is a subsequent set of priority actions. Here, the units first build up to 20 barracks towards the enemy's direction, if possible. Then, it prioritizes the harvest of resources by sending out six worker units. After resource collection, units then attack the less healthy enemy unit and finally, if no other actions are feasible, the units move towards the weakest ally unit for support or defense.
The overarching goal of this strategy is to bolster the player's resources and army strength before engaging in direct confrontation with the enemy. By keeping units idle initially, then focusing on resource collection and building a large army, the strategy aims to avoid premature engagement and instead build up a strong base and military power. Lastly, with a substantial force,

the strategy then aims to engage the enemy by targeting their weakest units first, utilizing the player's strength in numbers and resource advantage.

Best Reconstructed Program for P3:

```
for(Unit u) {
    u.idle();
    u.moveToUnit(Enemy, Closest);
    u.train(Worker, EnemyDir, 2);
    u.train(Heavy, EnemyDir, 50);
    u.train(Light, Left, 50);
    u.attack(Weakest);
    if(u.opponentHasUnitInPlayerRange()) {
        u.moveToUnit(Enemy, LessHealthy);
    }
    for(Unit u) {
        u.build(Barracks, EnemyDir, 20);
        u.harvest(6);
        u.attack(LessHealthy);
        if(u.canAttack()) {
            u.moveToUnit(Ally, Weakest);
        }
    }
}
```

P4:

```
for(Unit u){
    if(u.HasUnitInOpponentRange()) then {
        for(Unit u){
            u.moveToUnit(Enemy,Strongest)
        }
    }
    u.harvest(2)
    for(Unit u){
        for(Unit u){
            u.train(Light,Up,1)
        }
        u.train(Ranged,Left,50)
        u.idle()
    }
    u.build(Barracks,Down,20)
    u.train(Worker,EnemyDir,20)
```

```
    u.attack(Closest)
}
```

Explanation:

The strategy here revolves around specific actions prioritized to execute in order, facilitating effective gameplay in MicroRTS. To start, the utmost priority is given to the condition where our units are within an opponent's attack range. In this scenario, all units are commanded to move towards the enemy's strongest unit. This move helps to concentrate the attack on the enemy's most powerful unit, potentially destabilizing their strategies.
Subsequently, the strategy allows a couple of units to harvest resources. This step ensures we have a continuous supply of resources necessary for building structures and training units.
Next in the line of action priority, the strategy instructs all units to train a single Light unit upwards. This step is followed by training 50 Ranged units towards the left and then keeping them idle until an opponent comes within attack range. The idea here is to amass a significant number of Ranged units to bolster defenses, while the Light unit acts as a quick response unit to potential threats.
In the following step, the strategy commands to build up to 20 Barracks downwards from our current position. Barracks serve as the crucial point for training combat units. Hence, building Barracks broadens our ability to train more combat units, therefore strengthening our defenses and attack capabilities.
Subsequent to building Barracks, the strategy focuses on strengthening our workforce by training up to 20 Worker units towards the enemy's direction. The workers play a pivotal role in collecting resources and building structures, thus maintaining a strong workforce is vital to support our ongoing efforts in the game.
Finally, with the least priority, all units are directed to attack the closest opponent. This action ensures we maintain some pressure on the opponent even as we build up our forces and defenses.
The overall goal of this strategy is to establish strong defensive structures, maintain a robust workforce for resource collection and structure building, while continually training combat units for offensive and defensive purposes. All this is done while keeping an eye on the opponent's actions, enabling us to react swiftly to any potential threats.

Best Reconstructed Program for P4:

```
for(Unit u) {
    if(u.hasUnitInOpponentRange()) {
```

```
        u.moveToUnit(Enemy, Strongest)
    }

    for(Unit u) {
        u.harvest(2)
    }

    for(Unit u) {
        u.train(Light, Up, 1)
    }

    for(Unit u) {
        u.train(Ranged, Left, 50)
        u.idle()
    }

    for(Unit u) {
        u.build(Barracks, Down, 20)
    }

    for(Unit u) {
        u.train(Worker, EnemyDir, 20)
    }

    for(Unit u) {
        u.attack(Closest)
    }
}
```

P5:

```
for(Unit u){
    for(Unit u){
        u.train(Worker,EnemyDir,8)
        u.idle()
        for(Unit u){
            u.train(Light,Left,20)
        }
    }
    for(Unit u){
        u.harvest(2)
    }
    u.build(Barracks,Down,6)
```

```
    u.attack(Closest)
    u.train(Worker,Down,20)
}
```

Explanation:

The strategy provided is meant to be aggressive and prioritizes building a robust workforce first to support rapid unit production and resource gathering. The first and most crucial aspect of the strategy is to consistently and actively train workers and assign them to idle tasks. The idle state commands the workers to stay put and only attack if an enemy unit comes within their attack range. The strategy primarily focuses on training up to eight workers in the direction of the enemy. This action has the highest priority in our gameplay.
Simultaneously, there is also an emphasis on training up to 20 Light combat units positioned towards the left of the base or barracks. This task is of secondary importance in this strategy. Light units are typically fast and can attack well but may not be as durable. Hence, they can provide a good offensive line against the enemy.
Next, the strategy focuses on gathering resources by having units harvest them. Up to two units are chosen for this task at any given time. Although this action isn't the highest priority, it is crucial to ensure a steady inflow of resources for training more units and building more structures.
After the workforce and resources have been established, the next priority is to build up to six Barracks downwards from the base. Barracks are essential as they are the structures that train combat units, which are vital for both defense and offense in the game.
At the same time, the strategy dictates that our units should engage in battle with the closest enemy unit. This approach can ensure that any nearby threats are dealt with quickly.
The final action of this strategy is to train an additional set of up to 20 workers in the downward direction. While this action has the lowest priority, it ensures that we have enough workers in the late game to maintain production and resource gathering capabilities.
In summary, the overall goal of this strategy is to establish a robust workforce, ensuring a steady flow of resources, and maintaining a steady line of defense and offense against the enemy. The focus is on aggressive expansion and keeping pressure on the enemy while maintaining a robust economy.

Best Reconstructed Program for P5:

```
for(Unit u) {
    u.train(Worker, EnemyDir, 8)
```

```
    u.train(Light, Left, 20)
    u.harvest(2)
}

for(Unit u) {
    u.build(Barracks, Down, 6)
    u.attack(Closest)
}

for(Unit u) {
    u.train(Worker, Down, 20)
}
```

P6:

```
for(Unit u){
    for(Unit u){
        u.idle()
        u.harvest(2)
        for(Unit u){
            u.train(Light,Up,25)
        }
        for(Unit u){
            u.train(Worker,Up,2)
        }
        u.build(Barracks,Up,7)
    }
    u.attack(Closest)
    u.moveToUnit(Enemy,MostHealthy)
    u.train(Worker,EnemyDir,9)
}
```

Explanation:

In the context of the MicroRTS game, the strategy implied by this program prioritizes several actions in a specific order to achieve victory over the opponent. At the very outset, the highest priority is given to maintaining a defensive stance and gathering resources. So, the first thing all units, including Workers, Combat units (Light, Heavy, Ranged) will do is adopt a passive stance and wait for an enemy to approach them. In tandem, they will also focus on collecting resources, but with a limit of collecting from at most 2 resource units.
Once this defensive and resource gathering operation is in place, the game

strategy shifts to training new units, with a higher emphasis on Light combat units. Specifically, the next priority is to train as many as 25 Light units that will serve as the primary fighting force of the player's army. This action is followed closely by training of Worker units, with a cap of 2 new Workers to join the existing workforce for more efficient resource collection.
The next action in priority is to bolster the production capabilities by constructing Barracks, aiming to have up to 7 of them. This will allow faster training of new combat units, ensuring a steady stream of reinforcements.
After these initial preparations are complete, the strategy moves to more aggressive actions. Each unit in the player's force will focus on attacking the closest enemy unit. This proactive action is designed to minimize potential damage by eliminating nearby threats.
Following the attack, the next priority action is to maneuver our units strategically towards the healthiest enemy unit. This action can help in weakening the enemy's strong units and balancing the battlefield in favor of our army.
Lastly, and with the lowest priority in this strategy, the aim is to train additional Worker units, up to 9, near the enemy's direction. These Workers can help in speedy resource collection near enemy territory while also potentially serving as distractions for enemy attacks.
The overall goal of this strategy is to create a strong defense by prioritizing idle and resource collection actions, followed by expanding the army through training and construction of barracks. Once a strong defense and robust army are in place, the strategy switches to aggressive tactics by prioritizing attack and movement towards the enemy's healthiest units. The endgame is to have a strong and well-equipped army that can tackle any challenge, supported by an efficient workforce collecting resources close to the enemy base.

Best Reconstructed Program for P6:

```
for(Unit u) {
        u.idle()
        u.harvest(2)
    }

    for(Unit u) {
        u.train(Light, EnemyDir, 25)
        u.train(Worker, EnemyDir, 2)
    }

    for(Unit u) {
        u.build(Barracks, EnemyDir, 7)
    }
```

```
    for(Unit u) {
        u.attack(Closest)
    }

    for(Unit u) {
        u.moveToUnit(Enemy, MostHealthy)
    }

    for(Unit u) {
        u.train(Worker, EnemyDir, 9)
    }
```

P7:

```
for(Unit u){
    u.attack(Closest)
    u.train(Worker,Up,2)
    for(Unit u){
        u.idle()
        u.harvest(2)
    }
    for(Unit u){
        u.build(Barracks,Right,15)
        u.moveToUnit(Enemy,MostHealthy)
        u.train(Light,EnemyDir,8)
        u.moveToUnit(Enemy,Closest)
        for(Unit u){
            u.train(Worker,EnemyDir,15)
        }
    }
}
```

Explanation:

The strategy is organized in a hierarchical way, according to the priority of actions. The highest priority in this strategy is given to launching an attack on the nearest opponent unit. This action is initiated as soon as the game starts, suggesting an aggressive opening.
The second priority is to train two Worker units in the upward direction. These Workers will play a crucial role in resource management and building of structures.
The third priority is to command all the units to remain idle until an opponent unit comes within their attack range. These idle units, thus, are primarily defensive units.

In addition to this, two Workers are sent to harvest resources. These are likely the same Workers that were trained in the upward direction earlier. Harvesting resources is crucial in maintaining an economy and supporting the build and training of units.
At the next level of priority, a significant part of the strategy is unfolded. Each unit is given a sequence of tasks - building Barracks in the right direction, with a cap of 15, moving towards the healthiest enemy unit, training 8 Light units in the direction of the enemy, and moving again towards the closest enemy unit.
The construction of Barracks ensures a regular supply of combat units. The movement of units towards the healthiest and then the closest enemy units suggests a strategy that focuses on taking down the strongest opponent units first.
The training of Light units specifically points towards a preference for quicker, lighter forces over slow, heavy ones. Light units are often used for quick attacks and to keep the enemy's attention away from the main actions.
The lowest priority action is training 15 additional Workers in the direction of the enemy. This implies that there will be a late-game focus on building and resource collection, possibly in preparation for a final massive assault or to provide an economic edge over the opponent.
The overall goal of this strategy appears to be creating a powerful initial assault with rapid unit training and attack on the enemy, while simultaneously building up a steady resource income for future maneuvers. The later stages of the game will focus on strengthening the economy and mass-producing units for a final assault or a dominant economic victory.

Best Reconstructed Program for P7:

```
for (Unit u) {
    u.attack(Closest);
    u.train(Worker, Up, 2);
    u.idle();
    u.harvest(2);
    u.build(Barracks, Right, 15);
    u.moveToUnit(Enemy, MostHealthy);
    for (Unit u) {
        u.train(Light, EnemyDir, 8);
        u.moveToUnit(Enemy, Closest);
        u.train(Worker, EnemyDir, 15);
    }
}
```

P8:

```
for(Unit u){
u.train(Ranged,Left,100)
if(u.canHarvest()) then {
u.attack(Strongest)
}
for(Unit u){
u.train(Worker,Right,7)
}
u.train(Worker,Up,15)
for(Unit u){
u.build(Barracks,EnemyDir,5)
}
for(Unit u){
u.idle()
u.harvest(4)
}
u.attack(Closest)
for(Unit u){
u.train(Heavy,EnemyDir,3)
}
u.moveToUnit(Ally,Strongest)
}
```

Explanation:

The strategy is organized in a hierarchical way, according to the priority of actions. The highest priority in this strategy is given to launching an attack on the nearest opponent unit. This action is initiated as soon as the game starts, suggesting an aggressive opening.
The second priority is to train two Worker units in the upward direction. These Workers will play a crucial role in resource management and building of structures.
The third priority is to command all the units to remain idle until an opponent unit comes within their attack range. These idle units, thus, are primarily defensive units.
In addition to this, two Workers are sent to harvest resources. These are likely the same Workers that were trained in the upward direction earlier. Harvesting resources is crucial in maintaining an economy and supporting the build and training of units.
At the next level of priority, a significant part of the strategy is unfolded. Each unit is given a sequence of tasks - building Barracks in the right direction,

with a cap of 15, moving towards the healthiest enemy unit, training 8 Light units in the direction of the enemy, and moving again towards the closest enemy unit.
The construction of Barracks ensures a regular supply of combat units. The movement of units towards the healthiest and then the closest enemy units suggests a strategy that focuses on taking down the strongest opponent units first.
The training of Light units specifically points towards a preference for quicker, lighter forces over slow, heavy ones. Light units are often used for quick attacks and to keep the enemy's attention away from the main actions.
The lowest priority action is training 15 additional Workers in the direction of the enemy. This implies that there will be a late-game focus on building and resource collection, possibly in preparation for a final massive assault or to provide an economic edge over the opponent.
The overall goal of this strategy appears to be creating a powerful initial assault with rapid unit training and attack on the enemy, while simultaneously building up a steady resource income for future maneuvers. The later stages of the game will focus on strengthening the economy and mass-producing units for a final assault or a dominant economic victory.

Best Reconstructed Program for P8:

```
for(Unit u) {
    u.train(Ranged, Left, 100)
    if(u.canHarvest()) then {
            u.attack(Strongest)
    }
    for(Unit u) {
        u.train(Worker, Right, 7)
        u.train(Worker, Up, 15)

        for(Unit u) {
            if(u.canAttack()) then {
            u.build(Barracks, EnemyDir, 5)
        }
            u.idle()
            u.harvest(4)
            u.attack(Closest)
            u.train(Heavy, EnemyDir, 3)
            u.moveToUnit(Ally, Strongest)
        }
    }
}
```

P9:

```
for(Unit u){
    u.harvest(9)
    u.idle()
    u.moveToUnit(Enemy,LessHealthy)
    u.train(Worker,EnemyDir,6)
    for(Unit u){
        u.harvest(2)
    }
    for(Unit u){
        u.idle()
    }
    for(Unit u){
        if(u.HasUnitInOpponentRange()) then {
            u.attack(Weakest)
            u.moveToUnit(Enemy,LessHealthy)
        }
    }
    for(Unit u){
        u.moveToUnit(Enemy,MostHealthy)
        u.moveToUnit(Ally,MostHealthy)
        for(Unit u){
            u.build(Barracks,Left,7)
            u.train(Light,EnemyDir,20)
        }
    }
}
```

Explanation:

This strategy is geared towards aggressive resource gathering and expansion, while maintaining a robust defense. At the outset, our primary focus is on harvesting resources, with an emphasis on up to nine units engaging in this activity. As our units are engaged in resource gathering, they're also prepared to spring into action and defend our territory if required.
Simultaneously, we strategically move our units closer to the less healthy units of our enemy, targeting the weaker ones first to maximize our offensive capabilities. This isn't the highest priority, but a consistent task that we aim to achieve.
Along with these activities, we look to train six additional worker units, stationed towards the direction of the enemy. This serves dual purposes - enhancing our resource gathering capability and establishing a front line of defense against the enemy.

As we continue to build our resources and defenses, we deploy two units for more focused harvesting. This is more specific compared to the earlier broad directive and enjoys a higher priority.
This is closely followed by an order to have all our units maintain a vigilant stance, staying idle until there's a direct threat. This is a key element of our defense strategy.
A significant part of our strategy is to be proactively defensive. So, if any of our units are within the attacking range of the enemy, we go on the offensive by attacking the weakest units first, while strategically moving our units towards the less healthy enemy units. This has a higher priority and occurs before any other offensive or defensive moves.
The strategy also involves moving our units towards the most robust enemy and ally units. The aim here is to provide support to our strong units while simultaneously challenging the enemy's strong units.
To bolster our offense and defense, we aim to construct up to seven Barracks to the left of our existing base. Along with this, we train a massive force of twenty light combat units stationed towards the enemy. The building of the Barracks and the training of light units are part of our strategic expansion and are given a higher priority, falling just short of the highest priority.
Overall, this strategy is a blend of aggressive resource gathering, strategic defense, and tactical offense. Our goal is to build a strong resource base, maintain a defensive posture to thwart enemy attacks, and go on the offensive when the opportunity arises, aiming for a decisive victory.

Best Reconstructed Program for P9:

```
for (Unit u) {
    u.harvest(9);
    u.idle();
    u.moveToUnit(Enemy, LessHealthy);
    u.train(Worker, EnemyDir, 6);

    for (Unit u) {
        u.harvest(2);
        u.idle();
        if(u.hasUnitInOpponentRange()) {
            u.attack(Weakest);
        } else {
            u.moveToUnit(Enemy, LessHealthy);
        }
    }

    for (Unit u) {
```

```
        u.moveToUnit(Enemy, MostHealthy);
        u.moveToUnit(Ally, MostHealthy);
        u.build(Barracks, Left, 7);
        u.train(Light, EnemyDir, 20);
    }
}
```

P10:

```
for(Unit u){
    for(Unit u){
        u.train(Worker,Up,2)
    }
    u.idle()
    u.train(Heavy,EnemyDir,8)
}
for(Unit u){
    u.train(Light,Left,100)
    u.build(Barracks,EnemyDir,1)
    u.harvest(25)
    u.attack(Closest)
}
```

Explanation:

The provided strategy for playing MicroRTS consists of a series of prioritized actions. In this strategy, the most crucial action, with the highest priority, is to train worker units. Each of the controlled units is tasked with this function, effectively expanding the player's workforce as a first step. This action is prioritized by having the units train additional worker units in the upward direction, with the limit being up to 2 workers per unit. The second priority, which is slightly lower than the first one, is to ensure that the units stay idle when they are not training workers. In other words, until a unit has another specific task, it is required to stay idle, being ready to attack if any opponent comes within its attack range.
Next, there's the task of training heavy units. The units are given this task after they've finished training workers and become idle. Here, the strategy is to train up to 8 heavy units in the direction of the enemy, providing robust resistance against potential attacks.
Further down the priority list is the training of light units. Each unit is instructed to train light units to the left direction with a high limit of up to 100 units. This helps in preparing a strong army capable of handling various combat situations.

Building barracks in the enemy direction is also part of the strategy but with a lower priority than training units. Each unit is tasked to build up to one barracks in the direction of the enemy. This strategic placement could potentially disrupt the enemy's actions or serve as a forward base for our attacks.
Another crucial, yet lower priority task, is resource gathering. The strategy dictates that up to 25 workers are assigned to harvest resources. This provides the necessary resources to support the construction of buildings and training of additional units.
The last priority, with the lowest precedence, is to launch attacks on the enemy. Specifically, the units are ordered to attack the closest enemy unit. This offensive move is set with the lowest priority, implying that it should be executed only after all other tasks have been accomplished.
In summary, the strategy aims to expand and strengthen the player's forces through the training of different unit types, build a strategic forward base by erecting barracks near the enemy, manage resources by dedicating workers to harvesting, and finally, carry out attacks on the closest enemy units.

Best Reconstructed Program for P10:

```
for(Unit u) {
        u.train(Worker, Up, 2)
        u.idle()
        for(Unit u) {
            u.train(Heavy, EnemyDir, 8)
            u.train(Light, Left, 100)
            u.build(Barracks, EnemyDir, 1)
            u.harvest(25)
            u.attack(Closest)
        }
}
```

P11:

```
for(Unit u){
    u.train(Worker,Up,4)
}
for(Unit u){
    u.idle()
}
for(Unit u){
    for(Unit u){
        u.harvest(1)
```

```
    }
    u.train(Worker,Down,6)
    for(Unit u){
        u.train(Heavy,Left,10)
        for(Unit u){
            u.harvest(3)
        }
        u.build(Barracks,Left,15)
        u.attack(Closest)
    }
}
for(Unit u){
    u.train(Light,Left,100)
}
```

Explanation:

The program dictates a strategy for playing MicroRTS that emphasizes unit training and resource gathering. The action priorities are as follows: The highest priority action is to train Worker units at the top of your base. The game prioritizes the training of up to four Worker units. These units are crucial as they are responsible for gathering resources, building structures and even attacking opponent units.
Next in line, the strategy is to keep all units idle. This means that all units, including the previously trained Workers, will not be carrying out any specific task. They will only attack if an enemy unit comes within their attack range. The next set of actions revolve around both training more Worker units and collecting resources. With a high priority, every unit in the game will be assigned to harvest resources, but only one resource per unit. Then, up to six additional Worker units will be trained at the bottom of your base.
Following this, the strategy takes a more aggressive turn. The priority then shifts to training up to ten Heavy units on the left of your base. Heavy units, as the name suggests, are crucial in attacking and defending against the opponent units due to their increased survivability and damage. Meanwhile, within this training routine, there is a sub-priority to have each of these units harvest three resources. After training these Heavy units and their brief stint in resource gathering, up to fifteen Barracks will be constructed to the left of your base. Barracks are important as they allow the training of more combat units. This phase concludes with each unit attacking the closest enemy unit, signifying the initiation of an offensive against the opponent.
Lastly, the game ends with training up to a whopping 100 Light units on the left of your base. Light units, while weaker than Heavy units, can still provide a strong force in numbers and this shows the game's shift to a more aggressive approach.

The overall goal of this strategy appears to be a balanced approach of building up the base with Worker units and Barracks, collecting resources, and eventually moving towards an aggressive offense using Heavy and Light units.

Best Reconstructed Program for P11:

```
for(Unit u) {
    u.train(Worker,Up,4)
    u.idle()
    for(Unit u) {
        u.harvest(1)
        u.train(Worker,Down,6)
    }
    for(Unit u) {
        u.train(Heavy,Left,10)
        u.harvest(3)
        u.build(Barracks,Left,15)
        u.attack(Closest)
    }
    for(Unit u) {
        u.train(Light,Left,100)
    }
}
```

P12:

```
for(Unit u){
    u.train(Heavy,EnemyDir,6)
    u.train(Light,EnemyDir,4)
    u.build(Barracks,Down,3)
    u.idle()
    u.train(Worker,Left,3)
}
for(Unit u){
    for(Unit u){
        u.harvest(15)
    }
    for(Unit u){
        u.moveToUnit(Enemy,Weakest)
    }
}
```

Explanation:

In this MicroRTS strategy, the overall goal is to create a strong and sustainable army while harvesting resources efficiently. This strategy involves training and building activities along with harvesting and movement actions. In the first stage of this strategy, every unit is given a specific task, which is crucial to the early growth and power of our side. The highest priority is given to training our Heavy units in the direction of the enemy, aiming to produce six of them. After this, the strategy prioritizes training four Light units, again in the direction of the enemy. These units will provide initial offensive capabilities to confront any immediate enemy threat.
The third priority action is to build three Barracks in the downward direction, creating a stronghold to prepare for future engagements. Next, our units are commanded to idle, indicating they should hold their position and attack any opponent units that enter their range. Finally, in the first stage, our strategy ensures sustainability by training three Worker units in the left direction. These workers will be vital for resource collection and base expansion.
In the second stage, the strategy focuses on resource gathering and unit movement. The topmost priority at this stage is for every unit to harvest resources until we have 15 resources. This sustains our army and ensures that we can continue to build and train units.
Subsequently, each unit is instructed to move towards the weakest enemy unit. This targeted approach allows our forces to systematically weaken the enemy's defense line by focusing on their most vulnerable points.
The execution of these tasks in the given order of priority ensures that our army is well-prepared and resourceful, which is crucial for victory in MicroRTS.

Best Reconstructed Program for P12:

```
for(Unit u) {
    u.train(Heavy, EnemyDir, 6)
    u.train(Light, EnemyDir, 4)
    u.build(Barracks, Down, 3)
    u.idle()
    u.train(Worker, Left, 3)
}

for(Unit u) {
    u.harvest(15)
    u.moveToUnit(Enemy, Weakest)
}
```

P13:

```
for(Unit u){
    if(u.HasUnitWithinDistanceFromOpponent(5)) then {
        u.moveToUnit(Enemy,MostHealthy)
        u.moveToUnit(Ally,Farthest)
        u.attack(LessHealthy)
        u.train(Worker,Up,7)
    }
    u.attack(Weakest)
    for(Unit u){
        for(Unit u){
            u.harvest(1)
        }
        u.train(Light,Up,100)
        u.build(Barracks,EnemyDir,5)
        u.train(Worker,Left,4)
        u.idle()
        u.harvest(25)
    }
}
```

Explanation:

The first priority of the strategy is to check if any of our units are within 5 cells of an enemy unit. If this condition is met, the unit in question will carry out a set of actions. It will first move towards the healthiest enemy unit, indicating that our strategy is to target the strongest of the enemy. Immediately after that, the unit will also move towards our furthest ally unit. It is then commanded to attack the enemy unit with the least health. Following this, if the unit has the capability to train other units, it will train up to 7 workers in the upward direction. In case none of our units are within 5 cells of an enemy unit, the units will target the weakest enemy unit, regardless of its type or location. The second priority in the strategy is to instruct each unit to perform a number of actions. This starts with each unit being instructed to harvest one resource unit. This may be viewed as a micro-management strategy, to ensure that resources are constantly being gathered. Following this, each unit is instructed to train up to a maximum of 100 light units in the upward direction. Then, they are tasked to build up to 5 Barracks towards the enemy direction. If possible, the units are also instructed to train up to 4 workers towards the left direction. If a unit is unable to perform any of these tasks, it is ordered to remain idle and will only engage in combat if an enemy unit comes within its attack range. In addition to this, each unit is also tasked to harvest up to 25 resource units if possible.

In summary, the strategy seems to prioritize offensive actions towards the enemy's healthiest units while at the same time reinforcing our own position by training new units and building new Barracks. The strategy also emphasizes resource gathering by setting specific harvest actions for each unit. Thus, the overall goal of this strategy is to wear down the enemy's strongest units while simultaneously growing our own forces and resource pool.

Best Reconstructed Program for P13:

```
for(Unit u) {
    if(u.hasUnitWithinDistanceFromOpponent(5)) {
        u.moveToUnit(Enemy, MostHealthy);
        u.moveToUnit(Ally, Farthest);
        u.attack(LessHealthy);
        if(u.hasNumberOfUnits(Base, 1)) {
            u.train(Worker, Down, 7);
        }
    }
    else {
        u.attack(Weakest);
    }

    for(Unit u) {
        if(u.canHarvest()) {
            u.harvest(1);
        }
        else {
            u.train(Light, Up, 100);
            u.build(Barracks, EnemyDir, 5);
        }
        if(u.hasNumberOfUnits(Base, 1)) {
            u.train(Worker, Left, 4);
            u.idle();
        }
        else {
            u.harvest(25);
        }
    }
}
```

P14:

```
for(Unit u){
    u.idle()
    u.train(Light,EnemyDir,4)
    u.build(Barracks,Down,3)
    u.train(Ranged,EnemyDir,8)
    u.train(Worker,Left,3)
    u.harvest(15)
}
for(Unit u){
    for(Unit u){
        for(Unit u){
            u.attack(Closest)
        }
    }
}
```

Explanation:

In the strategy described, units initially take on an idle state, poised to attack any opponent unit that comes within range. This marks the first course of action. Following that, a total of four units are then trained for light combat, oriented towards the enemy's direction. This is the second action taken. After preparing for combat, our focus shifts towards infrastructure development, with the construction of three barracks in the downward direction. This is the third action in line. Post the infrastructure setup, we further train eight ranged combat units, pointing again towards enemy territory, forming the fourth action.
In addition to training combat units, we also train three worker units, placed on the left, which makes up the fifth action. These worker units are immediately assigned to gather resources with a target of harvesting up to fifteen resource units, and this is the sixth action our strategy involves.
The strategy then shifts gears and goes into an aggressive mode. All units, regardless of their type, are assigned to attack the enemy's closest unit. This forms the most crucial and high-priority action in this plan, making it the top priority. This order is repeated for every single one of our units, ensuring a relentless wave of attacks aimed at the nearest enemy units.
Overall, this strategy involves starting with a defensive stance and resource accumulation, followed by a massive, prioritized attack on the closest enemy units, signifying an aggressive playstyle.

Best Reconstructed Program for P14:

```
for(Unit u) {
        u.idle();
        u.train(Light, EnemyDir, 4);
        u.build(Barracks, Down, 3);
        u.train(Ranged, EnemyDir, 8);
        u.train(Worker, Left, 3);
        u.harvest(15);
}
for(Unit u) {
    u.attack(Closest);
}
```

P15:

```
for(Unit u){
    u.train(Worker,EnemyDir,3)
    u.build(Barracks,Up,1)
    u.moveToUnit(Enemy,Closest)
    u.attack(MostHealthy)
    u.moveToUnit(Ally,MostHealthy)
    for(Unit u){
        u.idle()
    }
    for(Unit u){
        u.train(Light,EnemyDir,15)
    }
    for(Unit u){
        u.harvest(2)
    }
}
```

Explanation:

In the strategy we're discussing, there's a clear order of priorities to the actions our units will take. At the very top of our priority list, the main thing units are encouraged to do is to stand their ground and defend their position. If you see a unit doing nothing, that's because they've been instructed to remain idle as their highest priority.
Following that, units have a higher priority to train Light combat units in the direction of the enemy. This means our main offensive strategy focuses on producing as many as 15 Light units to apply pressure on the opponent.
The next action with a somewhat lower priority than training Light units is resource gathering. Here, the directive is to send units out to harvest resources. The goal is to send up to 2 Workers to gather resources at any

given time.
After those high-priority tasks, if none of those directives apply, our units then proceed with the next set of actions. The first among these is to train Workers, with an emphasis on training up to 3 of them in the direction of the enemy. This ensures that our army has a steady supply of units that can gather resources and build structures.
Once we have a sufficient number of Workers, the next objective is to build a Barracks in the upward direction. However, we limit this construction to just 1 Barracks. With a Barracks in place, we can then train combat units to bolster our army's strength.
With training and building out of the way, our units are then focused on movement and engagement strategies. They are instructed to approach the closest enemy unit and engage. Specifically, they will target the healthiest opponent unit, aiming to take down the strongest threats first.
In situations where our units need support, they are encouraged to move towards our most healthy ally unit. This could be a tactical move to bolster defense or provide a more united front in our offensive maneuvers.
Overall, the central aim of this strategy seems to be a balanced approach between defense, resource management, and offensive pressure. By prioritizing idle defense and the training of Light units, it suggests a reactive posture that is ready to quickly switch to offense by amassing Light units when the situation allows. The emphasis on resource gathering and Worker training ensures that the army is well-supported and can sustain prolonged engagements

Best Reconstructed Program for P16:

```
for(Unit u) {
    u.train(Worker, EnemyDir, 3)
    u.build(Barracks, Up, 1)
    u.moveToUnit(Enemy, Closest)
    u.attack(MostHealthy)
    u.moveToUnit(Ally, MostHealthy)

    for(Unit u) {
        u.idle()
    }

    for(Unit u) {
        u.train(Light, EnemyDir, 15)
        u.harvest(2)
    }
}
```

P16:

```
for(Unit u){
    u.harvest(2)
    u.idle()
    u.train(Light,EnemyDir,7)
    u.build(Barracks,EnemyDir,3)
    u.train(Ranged,EnemyDir,8)
    u.train(Worker,Left,3)
    u.moveToUnit(Enemy,MostHealthy)
    u.moveAway()
}
for(Unit u){
    u.harvest(50)
    u.attack(Strongest)
    if(u.HasUnitWithinDistanceFromOpponent(3)) then {
        u.train(Worker,Right,10)
    }
}
```

Explanation:

In this MicroRTS strategy, our units prioritize their actions based on the following order:
1. First and foremost, they'll start by harvesting resources, but they will only send two units for this task. 2. If they aren't harvesting, they'll stay idle and defend if an opponent comes within their attack range. 3. If they haven't been assigned any of the above tasks, they'll then look to train Light combat units, specifically focusing on producing up to seven of them, directing them towards the enemy. 4. As they look to fortify their position, they'll then consider constructing Barracks. The units will try to build up to three Barracks in the direction of the enemy. 5. If their earlier training and building tasks are satisfied, they'll pivot to training Ranged units, aiming to have eight of these units, and they'll also be sent towards the enemy. 6. To ensure continuous resource collection and building, they'll train up to three Workers who will be positioned on the left side. 7. After all these tasks, if none of them are actionable for a unit, it will look for the healthiest opponent unit and move towards it. 8. Finally, if a unit hasn't been assigned any task till now, it'll choose to move away from the player's base.
Once all units have gone through the above priorities, they'll move to the next set of actions:
1. An intense resource collection activity will ensue, with a large group of 50 units being sent out to harvest. 2. When not busy harvesting, they'll go on

the offensive, targeting the strongest opponent units to attack. 3. A special consideration is made for situations where they're close to the enemy. If a unit is within a distance of 3 from an opponent, it'll train up to ten Workers, positioning them on the right side.
The main goal of this strategy seems to be establishing a strong frontline with trained combat units and Barracks, while also ensuring there's a continuous supply of resources by having a large number of Workers. Moreover, it places a strong emphasis on being proactive in engagements by targeting stronger and healthier opponent units.

Best Reconstructed Program for P16:

```
for(Unit u) {
    for(Unit u) {
        if(u.canHarvest()) {
            u.harvest(2)
        }
        u.idle()
        u.train(Light, EnemyDir, 7)
        u.build(Barracks, EnemyDir, 3)
        u.train(Ranged, EnemyDir, 8)
        u.train(Worker, Left, 3)
        u.moveToUnit(Enemy, MostHealthy)
        u.moveAway()
    }
    for(Unit u) {
        u.harvest(50)
        u.attack(Strongest)
        if(u.hasUnitWithinDistanceFromOpponent(3)) {
            u.train(Worker, Right, 10)
        }
    }
}
```

P17:

```
for(Unit u){
    for(Unit u){
        u.idle()
    }
    u.train(Worker,EnemyDir,5)
    u.harvest(3)
    u.train(Heavy,Right,50)
```

```
    u.moveToUnit(Enemy,MostHealthy)
    u.moveAway()
    if(u.OpponentHasUnitInPlayerRange()) then {
        for(Unit u){
            u.harvest(1)
            u.attack(Strongest)
        }
    }
    for(Unit u){
        u.build(Barracks,EnemyDir,1)
    }
}
```

Explanation:

The given strategy for playing MicroRTS presents a specific approach to handling the game, primarily focusing on certain aspects of controlling units and managing their actions.
The strategy commences with the highest priority action, where all units are commanded to stay idle and attack if an opponent's unit comes within attack range. This will form the core of the defense and is a priority to ensure that units are initially in a guarded state.
Next, the actions following that are of second-highest priority: - The player's unit trains up to 5 Worker units in the direction of the enemy. - 3 Worker units are sent to harvest resources. - 50 Heavy combat units are trained to the right. - Units are commanded to move towards the enemy's most healthy unit. - Units are commanded to move in the opposite direction of the player's base.
These actions create a balance between building resources, strengthening the army, and engaging the enemy strategically.
The third-highest priority action is an immediate response mechanism. If an opponent's unit is within the attack range of an ally's unit, the player's units are commanded to: - Harvest one resource each. - Attack the strongest opponent unit.
This section forms a rapid reaction to an immediate threat, emphasizing attack and gaining resources in parallel.
Lastly, the action with the lowest priority is to build a Barracks in the direction of the enemy, limited to just one Barracks. This is the least urgent aspect of the strategy, focusing on long-term building rather than immediate response or unit training.
Overall, the strategy focuses on an immediate defensive stance with a systematic escalation in attacking, resource building, and strategic movement. The ultimate goal is to prepare the units quickly for both defense and attack,

with a layered approach to handling threats, building resources, and then extending the infrastructure by constructing a Barracks. The priority is set in such a way that immediate defense and controlled aggression take precedence over the long-term development of structures.

Best Reconstructed Program for P17:

```
for(Unit u) {
    u.idle();

    for(Unit u) {
        u.train(Worker, EnemyDir, 5);
        u.harvest(3);
        u.train(Heavy, Right, 50);
        u.moveToUnit(Enemy, MostHealthy);
        u.moveAway();
    }

    for(Unit u) {
        if(u.opponentHasUnitInPlayerRange()) {
            u.harvest(1);
            u.attack(Strongest);
        }
    }

    u.build(Barracks, EnemyDir, 1);
}
```

P18:

```
for(Unit u){
    u.idle()
    u.train(Light,Right,10)
    if(u.OpponentHasUnitInPlayerRange()) then {
        u.moveToUnit(Ally,Strongest)
    } else {
        u.train(Worker,EnemyDir,7)
    }
    u.build(Barracks,Down,20)
    u.moveToUnit(Enemy,Closest)
    for(Unit u){
        u.train(Worker,Right,5)
    }
```

```
    if(u.canAttack()) then {
        u.attack(Weakest)
    } else {
        e
    }
    for(Unit u){
        if(u.canHarvest()) then {
            u.harvest(2)
        }
        u.build(Barracks,EnemyDir,8)
    }
}
```

Explanation:

The given strategy for playing MicroRTS focuses on a balanced approach to both defense and offense, prioritizing certain actions to achieve desired outcomes. First, the strategy commands all units to stay idle, which is of lower priority. If they are able to, they will then train 10 Light units to the right. The strategy then assesses the battlefield: if an opponent unit is in range of an ally unit, the priority is given to moving units towards the strongest Ally unit; otherwise, the command to train 7 Worker units towards the enemy direction is executed.
After assessing the battlefield, the priority shifts to building. Up to 20 Barracks are built downwards. This is followed by a command to move towards the closest enemy, which is of lower priority compared to previous actions.
Next, the strategy enters a phase where the priority is the highest, focusing on training and preparing for the battles ahead. In this phase, up to 5 Worker units are trained to the right.
Following this, the strategy again checks if the units can attack. If they can, they are commanded to attack the weakest opponent unit. If not, no action is taken.
In the final phase, which has the second-highest priority, the focus is on harvesting resources and further building. If units can harvest, they are instructed to harvest 2 resources. Additionally, up to 8 Barracks are built towards the enemy direction.
Overall, the strategy aims for a balanced approach. It prioritizes training and preparing for battle through building Barracks and training various types of units. It takes into account the positioning of both ally and enemy units, emphasizing moving and attacking based on the enemy's position and strength. It also focuses on resource management by training Workers and commanding them to harvest. The highest priority actions are training Workers, followed by building Barracks, attacking or defending based on the opponent's positioning, and finally, commanding units to move, stay idle, or

train additional Light units.

Best Reconstructed Program for P18:

```
for (Unit u) {
    u.idle()
    if (u.canAttack()) {
        u.attack(Weakest)
    }
    u.build(Barracks, EnemyDir, 8)
    u.train(Worker, Right, 7)
    u.train(Light, Right, 10)
    if (u.opponentHasUnitInPlayerRange()) {
        u.moveToUnit(Ally, Strongest)
    } else {
        u.train(Worker, EnemyDir, 7)
    }
    u.build(Barracks, Down, 20)
    u.moveToUnit(Enemy, Closest)
    for (Unit u) {
        u.train(Worker, Right, 5)
        if (u.canHarvest()) {
            u.harvest(2)
        }
        u.build(Barracks, EnemyDir, 8)
    }
}
```

P19:

```
for(Unit u){
    for(Unit u){
        u.idle()
        u.train(Light,Down,4)
    }
    for(Unit u){
        u.harvest(3)
        for(Unit u){
            u.train(Ranged,Down,20)
        }
    }
    u.build(Barracks,EnemyDir,50)
    u.attack(Closest)
```

```
    u.train(Worker,Down,8)
}
```

Explanation:

In this MicroRTS strategy, players will initially prioritize having their units stay idle and thereafter focus on training Light units. This action receives the highest priority. Once that's achieved, the player's next focal point is to ensure that up to 3 Worker units start harvesting resources. During this harvesting phase, there's also an emphasis on training up to 20 Ranged units. This step is of the second highest priority.
As the player progresses further, constructing Barracks in the direction of the enemy becomes a vital move. The game will aim to build up to 50 of these structures, suggesting a major strategic emphasis on increasing military capabilities. Following that, the units are directed to attack the closest enemy units. However, these actions aren't given as high a priority as the initial actions.
Lastly, with a slightly lower priority, the strategy emphasizes training additional Worker units. The player will try to bring about 8 more Workers into the game.
To sum it up, the overall goal of this strategy seems to be a balanced approach between defense (having units stay idle initially) and offense (training Light and Ranged units). Furthermore, there's a strong emphasis on resource gathering and expanding the military capacity by building Barracks near the enemy. The balanced focus between offense, defense, and resource gathering suggests a comprehensive approach to the game.

Best Reconstructed Program for P19:

```
for(Unit u) {
    u.idle()

    for(Unit u) {
        u.train(Light, Down, 4)
    }

    for(Unit u) {
        u.harvest(3)
        u.train(Ranged, Down, 20)
    }

    for(Unit u) {
        u.build(Barracks, EnemyDir, 50)
```

```
        u.attack(Closest)
        u.train(Worker, Down, 8)
    }
}
```

P20:

```
for(Unit u){
    for(Unit u){
        u.train(Heavy,EnemyDir,3)
    }
    for(Unit u){
        u.idle()
    }
    u.harvest(3)
    u.build(Base,EnemyDir,1)
    u.train(Light,Left,8)
    u.build(Barracks,Right,1)
    u.attack(Weakest)
    for(Unit u){
        u.train(Heavy,Up,10)
    }
    u.train(Worker,EnemyDir,4)
}
```

Explanation:

In this MicroRTS strategy, we initiate by placing the utmost importance on training Heavy units. The prime focus is to create three Heavy units positioned towards the enemy direction. This is the most crucial step of the strategy and receives the highest priority.
Following this, the units are instructed to maintain a stance of inaction, standing their ground. This takes precedence right after training the Heavy units.
As we move further down the list of priorities, the next action involves sending three Worker units to gather resources. Then, there's a directive to construct a Base, positioning it towards the enemy's direction. Only one Base will be built under this directive.
Subsequent to these actions, the strategy emphasizes creating a considerable force of Light units, specifically eight of them, positioned to the left. This is then followed by the establishment of a Barracks to the right, but only one will be erected.
The next phase of the strategy is to be more aggressive. Units are ordered to target and attack the weakest of the opponent's forces. This aggressive stance is vital before moving on to the subsequent steps.

Delving deeper, there's a strong inclination to further enhance our Heavy unit troops. This time, the strategy pushes for the creation of ten Heavy units, with their position being upwards.
Lastly, the strategy rounds off by emphasizing the training of four Worker units, positioning them towards the enemy's direction.
Overall, this strategy can be viewed as a balanced approach between fortifying defenses, resource management, and mounting an offensive. The primary goal is to strengthen the army by training a significant number of Heavy and Light units while ensuring resource accumulation and strategic positioning of bases and barracks.

Best Reconstructed Program for P20:

```
for (Unit u) {
    for (Unit u) {
        u.train(Heavy, EnemyDir, 3);
        u.idle();
    }
    u.harvest(3);
    u.build(Base, EnemyDir, 1);
    u.train(Light, Left, 8);
    u.build(Barracks, Right, 1);
    u.attack(Weakest);
    for (Unit u) {
        u.train(Heavy, Up, 10);
    }
    u.train(Worker, EnemyDir, 4);
}
```

B.6.2 Human-crafted Set

P1:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType() == baseType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
            int nworkers = 0;
            for (Unit u2 : pgs.getUnits()) {
                if (u2.getType() == workerType &&\
                u2.getPlayer() == p.getID()) {
                    nworkers++;
                }
            }
```

```
        if (nworkers < 1 && p.getResources() >= workerType.cost) {
            train(u, workerType);
        }
    }
}
```

Explanation:

In this strategy, the game checks for each unit on the game map. It focuses on the units that are of base type and belong to the player who doesn't have any assigned action. Once it finds such a base, it looks for all the worker units that belong to the player. It keeps a track of the number of such workers. If the player has less than one worker and has enough resources to train a worker, the base is commanded to train a worker unit. This strategy ensures that the player always has at least one worker unit if resources permit. The overall goal of this strategy is to maintain the number of worker units for the player, which is crucial for collecting resources and building structures.

Best Reconstructed Program for P1:

```
for(Unit u : pgs.getUnits()) {
    if (u.getType() == baseType && u.getPlayer() == player) {
        if (gs.getActionAssignment(u) == null) {
            int numWorkers = 0;
            for(Unit u2 : pgs.getUnits()) {
                if (u2.getType() == workerType &&\
                u2.getPlayer() == player) {
                    numWorkers++;
                }
            }
            if (numWorkers < 1 &&\\
            p.getResources() >= workerType.cost) {
                train(u, workerType);
            }
        }
    }
}
```

P2:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType() == barracksType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
```

```
        if (p.getResources() >= lightType.cost) {
            train(u, lightType);
        }
    }
}
```

Explanation:

This strategy focuses on training Light units from your Barracks. For every unit on the game map, the strategy checks if the unit is a Barracks and belongs to the player. The strategy then verifies if the Barracks is not currently assigned any action to perform. If the Barracks is idle and belongs to the player, the strategy looks at the amount of resources the player currently possesses. If the player has enough resources to train a Light unit (a Light unit's cost is the benchmark here), the strategy directs the Barracks to train a Light unit. The Light unit is a type of combat unit and can be useful in attacking the opponent's units. sThe overall goal of this strategy is to continuously train Light units from the Barracks whenever there are enough resources available and the Barracks is idle. This way, the player can build a strong army of Light units to take on the opponent.

Best Reconstructed Program for P2:

```
List<Unit> units = pgs.getUnits();
for(Unit u : units) {
  if(u.getType() == barracksType && u.getPlayer() == player) {
    if(gs.getActionAssignment(u) == null) {
      if(p.getResources() >= lightType.cost) {
        train(u, lightType);
      }
    }
  }
}
```

P3:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType().canAttack && !u.getType().canHarvest &&\
    u.getPlayer() == player && gs.getActionAssignment(u) == null) {
        PhysicalGameState pgs = gs.getPhysicalGameState();
        Unit closestEnemy = null;
        int closestDistance = 0;
        int mybase = 0;
```

```
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getPlayer() >= 0 && u2.getPlayer() != p.getID())
            {
                int d = Math.abs(u2.getX() - u.getX()) +\
                Math.abs(u2.getY() - u.getY());
                if (closestEnemy == null || d < closestDistance) {
                    closestEnemy = u2;
                    closestDistance = d;
                }
            }
            else if(u2.getPlayer()==p.getID() &&\
            u2.getType() == baseType){
                mybase = Math.abs(u2.getX() - u.getX()) +\
                Math.abs(u2.getY() - u.getY());
            }
        }
        if (closestEnemy!=null &&\
        (closestDistance < pgs.getHeight()/2  \
        || mybase < pgs.getHeight()/2)){
            attack(u,closestEnemy);
        }
        else
        {
            attack(u, null);
        }
    }
}
```

Explanation:

In the given strategy, the player controls specific units that can attack but cannot gather resources. The strategy involves these units seeking out and attacking enemy units within a certain range. The first step is to identify all units that can attack but not harvest, and are controlled by the player. For each of these units, the strategy determines its closest enemy. This is done by calculating the distance between the player's unit and each enemy unit on the game map, with distance being the sum of the differences in their X and Y coordinates. Simultaneously, for each of these player's units, the strategy determines the distance to the player's base. This is done by identifying all units that belong to the player and are of the base type, and calculating the distance between the player's unit and the base. Once the closest enemy and the distance to the base are determined, the player's unit is commanded to attack the enemy if the enemy is closer than half the height of the game map or if the base is closer than half the height of the game map. If neither of

these conditions is met, the unit is commanded to stop attacking. Overall, the goal of this strategy is to command units that can attack but cannot harvest to engage in combat with enemy units that are close by or pose a threat to the base. The strategy ensures that units are not sent too far from the base and are able to protect the base if it comes under attack.

Best Reconstructed Program for P3:

```
int halfHeight = pgs.getHeight() / 2;
List<Unit> units = pgs.getUnits();
for(Unit u : units) {
    if(u.getType().canAttack && !u.getType().canHarvest &&\
    u.getPlayer() == player) {
        Unit closestEnemy = null;
        Unit base = null;
        int closestDistance = Integer.MAX_VALUE;
        int baseDistance = Integer.MAX_VALUE;
        for(Unit u2 : units) {
            int dx = u2.getX() - u.getX();
            int dy = u2.getY() - u.getY();
            if(u2.getPlayer() >= 0 && u2.getPlayer() != player) {
                int distance = Math.abs(dx) + Math.abs(dy);
                if(distance < closestDistance) {
                    closestEnemy = u2;
                    closestDistance = distance;
                }
            }
            if(u2.getType() == baseType && u2.getPlayer() == player)
            {
                baseDistance = Math.abs(dx) + Math.abs(dy);
                base = u2;
            }
        }
        if(closestEnemy != null && \\
        (closestDistance < halfHeight || baseDistance < halfHeight))
        {
            attack(u, closestEnemy);
        }
        else {
            attack(u, null);
        }
    }
}
```

P4:

```
resourcesUsed=gs.getResourceUsage().getResourcesUsed(player);
for (Unit u : pgs.getUnits()) {
    if (u.getType() == baseType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
        int nworkers = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == workerType
                    && u2.getPlayer() == p.getID()) {
                nworkers++;
            }
        }
        int nBases = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == baseType
                    && u2.getPlayer() == p.getID()) {
                nBases++;
            }
        }
        int qtdWorkLim = nWorkerBase * nBases;
        if (nworkers < qtdWorkLim &&\
        p.getResources() >= workerType.cost) {
            train(u, workerType);
        }
    }
}
```

Explanation:

The strategy in this program is about managing the bases and workers in the game. It first checks all units on the map. If it finds a base that belongs to the player and is not currently assigned any action, it starts to execute the strategy. The strategy begins by counting the number of workers that the player currently has on the map. Then, it counts the number of bases that the player owns. Next, it calculates the limit of workers that should be assigned to each base. This limit is determined by the product of a predefined constant (nWorkerBase) and the number of bases that the player owns. If the number of workers is less than this limit and the player has enough resources to train a new worker, the base is instructed to train a worker. The overall goal of this strategy is to maintain an optimal number of workers for each base that the player owns, given the resources available. This is done to ensure that the player can collect resources efficiently in the game.

Best Reconstructed Program for P4:

```
List<Unit> units = pgs.getUnits();
for(Unit u : units) {
    if(u.getType() == baseType && u.getPlayer() == player) {
        if(gs.getActionAssignment(u) == null) {
            int numWorkers = 0;
            for(Unit u2 : units) {
                if(u2.getType() == workerType &&\
                u2.getPlayer() == player) {
                    numWorkers++;
                }
            }
            int numBases = 0;
            for(Unit u2 : units) {
                if(u2.getType() == baseType &&\
                u2.getPlayer() == player) {
                    numBases++;
                }
            }
            int workerLimit = numBases * nWorkerBase;
            if(numWorkers < workerLimit &&\
            p.getResources() >= workerType.cost) {
                // Instruct base to train a worker
                train(u, workerType);
            }
        }
    }
}
```

P5:

```
resourcesUsed=gs.getResourceUsage().getResourcesUsed(player);
for (Unit u : pgs.getUnits()) {
    if (u.getType() == barracksType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
        int nLight = 0;
        int nRanged = 0;
        int nHeavy = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == lightType
                    && u.getPlayer() == p.getID()) {
                nLight++;
            }
```

```
            if (u2.getType() == rangedType
                    && u.getPlayer() == p.getID()) {
                nRanged++;
            }
            if (u2.getType() == heavyType
                    && u.getPlayer() == p.getID()) {
                nHeavy++;
            }
        }
        if (nLight == 0 &&\
        p.getResources() >= (lightType.cost + resourcesUsed)) {
            train(u, lightType);
            resourcesUsed += lightType.cost;
        } else if (nRanged == 0 &&\
        p.getResources() >= (rangedType.cost + resourcesUsed)) {
            train(u, rangedType);
            resourcesUsed += rangedType.cost;
        } else if (nHeavy == 0 &&\
        p.getResources() >= (heavyType.cost + resourcesUsed)) {
            train(u, heavyType);
            resourcesUsed += heavyType.cost;
        }
        if (p.getResources() >= baseType.cost &&\
        nLight != 0 && nRanged != 0 && nHeavy != 0) {
            int number = r.nextInt(3);
            switch (number) {
                case 0:
                    if (p.getResources() >=
                    (baseType.cost+lightType.cost)) {
                        train(u, lightType);
                        resourcesUsed += lightType.cost;
                    }
                    break;
                case 1:
                    if (p.getResources() >=
                    (baseType.cost+rangedType.cost)) {
                        train(u, rangedType);
                        resourcesUsed += rangedType.cost;
                    }
                    break;
                case 2:
                    if (p.getResources() >=
                    (baseType.cost+ heavyType.cost)) {
```

```
                        train(u, heavyType);
                        resourcesUsed += heavyType.cost;
                    }
                    break;
                }
            }
        }
    }
```

Explanation:

The strategy described in the program involves managing the use of Barracks units and the resources of the player. The Barracks units are controlled by the player and are used to train combat units which are Light, Heavy, or Ranged. In the beginning, the program checks each unit within the player's control. If a unit is a Barracks unit, belongs to the player, and is not currently assigned an action, it proceeds to the next step. The strategy then counts the number of each type of combat unit (Light, Heavy, and Ranged) that the player currently has. It does this by going through each unit the player controls and checking if it's a Light, Heavy, or Ranged unit. Once the numbers of each type of combat unit are determined, the program checks if the player has any Light units. If not, and the player has enough resources to train a Light unit, it commands the Barracks to train a Light unit. If the player already has a Light unit, it checks for a Ranged unit and repeats the process. If the player has both Light and Ranged units, it checks for a Heavy unit and repeats the process again. The above process ensures that the player has at least one of each type of combat unit if they have enough resources. However, if the player already has all types of units and still has enough resources to train another unit plus build a base, it randomly selects one type of unit (Light, Heavy, or Ranged) to train. The overall goal of the strategy is to ensure the player has a balanced mix of combat units (Light, Heavy, and Ranged) and to use the available resources efficiently for training additional units or building a base.

Best Reconstructed Program for P5:

```
for(Unit u: pgs.getUnits()) {
    if (u.getType() == barracksType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
        int numLight = 0;
        int numHeavy = 0;
        int numRanged = 0;
        for(Unit u2: pgs.getUnits()) {
```

```
            if (u2.getType() == lightType &&\
            u2.getPlayer() == player) {
                numLight++;
            } else if (u2.getType() == heavyType &&\
            u2.getPlayer() == player) {
                numHeavy++;
            } else if (u2.getType() == rangedType &&\
            u2.getPlayer() == player) {
                numRanged++;
            }
        }
        if (numLight == 0 &&\
        p.getResources() >= lightType.cost) {
            train(u, lightType);
        }
        else if (numRanged == 0 &&\
        p.getResources() >= rangedType.cost) {
            train(u, rangedType);
        }
        else if (numHeavy == 0 &&\
        p.getResources() >= heavyType.cost) {
            train(u, heavyType);
        }
        else if (p.getResources() >= lightType.cost + baseType.cost)
        {
            int randomUnit = (int)(Math.random() * 3);
            if (randomUnit == 0) {
                train(u, lightType);
            } else if (randomUnit == 1) {
                train(u, heavyType);
            } else {
                train(u, rangedType);
            }
        }
    }
}
```

P6:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType().canAttack && !u.getType().canHarvest &&\
    u.getPlayer() == player && gs.getActionAssignment(u) == null) {
        PhysicalGameState pgs = gs.getPhysicalGameState();
```

```
        Unit closestEnemy = null;
        int closestDistance = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getPlayer() >= 0 && u2.getPlayer() != p.getID())
            {
                int d = Math.abs(u2.getX() - u.getX()) +\
                Math.abs(u2.getY() - u.getY());
                if (closestEnemy == null || d < closestDistance) {
                    closestEnemy = u2;
                    closestDistance = d;
                }
            }
        }
        if (closestEnemy != null) {
            attack(u, closestEnemy);
        }
    }
}
```

Explanation:

This strategy is about using the combat units of the player for attacking the enemy units. Initially, the strategy looks at each unit in the game. It then considers those units which can attack, cannot harvest and belong to the player. Moreover, it only considers those units which are currently not assigned any other action. For each such combat unit, the strategy identifies the closest enemy unit. It calculates the distance between the combat unit and all units that belong to the opponent. The distance is calculated as the sum of the absolute differences of the x and y coordinates of the two units. The unit with the minimum distance is considered as the closest enemy. Once the closest enemy is found for a combat unit, it commands the combat unit to attack the closest enemy unit. The overall goal of the strategy is to engage the player's combat units in attacking the nearest enemy units, to potentially eliminate the enemy forces.

Best Reconstructed Program for P6:

```
for(Unit u : pgs.getUnits()) {
        if(u.getPlayer() == player && u.getType().canAttack &&\
        !u.getType().canHarvest) {
            if(gs.getActionAssignment(u) == null) {
                int minDistance = Integer.MAX_VALUE;
                Unit closestEnemy = null;
                for(Unit u2 : pgs.getUnits()) {
```

```
                    if(u2.getPlayer() >= 0 &&\
                    u2.getPlayer() != player) {
                        int dx = u2.getX() - u.getX();
                        int dy = u2.getY() - u.getY();
                        int distance = Math.abs(dx) + Math.abs(dy);
                        if(distance < minDistance) {
                            minDistance = distance;
                            closestEnemy = u2;
                        }
                    }
                }
                if (closestEnemy != null) {
                    attack(u, closestEnemy);
                }
            }
        }
    }
```

P7:

```
List<Integer> reservedPositions = new LinkedList<>();
for (Unit u : pgs.getUnits()) {
    if (u.getType() == baseType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
        int nworkers = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == workerType
                    && u2.getPlayer() == p.getID()) {
                nworkers++;
            }
        }
        int nBases = 0;
        int nBarracks = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == baseType
                    && u2.getPlayer() == p.getID()) {
                nBases++;
            } else if (u2.getType() == barracksType
                    && u2.getPlayer() == p.getID()) {
                nBarracks++;
            }
        }
        int qtdWorkLim;
```

```
        if(nBarracks == 0){
            qtdWorkLim = 4;
        }else{
            qtdWorkLim = nWorkerBase * nBases;
        }
        if (nworkers < qtdWorkLim &&\
        p.getResources() >= workerType.cost) {
            train(u, workerType);
        }
    }
}
```

Explanation:

In this strategy, the player controls units in the MicroRTS game. Initially, the strategy checks if a base unit controlled by the player is not currently assigned any action. This is important because a unit can only perform one action at a time. Once it finds such a base, the strategy begins by counting the number of worker units controlled by the player. Workers are essential units that collect resources and build structures which are crucial for the advancement of the game. After counting the workers, the strategy proceeds to count the number of base and barracks units controlled by the player. Bases are where workers are trained, while barracks are where combat units are trained. Next, the strategy determines the limit of workers that can be trained. If there are no barracks, the limit is set to 4. However, if there are barracks, the limit is set to the number of workers per base times the number of bases. This decision is made based on the fact that having more barracks means the player is more likely to be in a combat-intensive situation and would require more resources, thus needing more workers. Finally, if the number of workers is less than the worker limit and the player has enough resources to train a worker (workers come at a cost), the base is commanded to train a worker. The overall goal of this strategy is to make sure that the player has an optimal number of workers based on the current state of the game. This is crucial because having an appropriate number of workers can ensure a steady flow of resources, which can contribute to the player's success in the game.

Best Reconstructed Program for P7:

```
List<Unit> units = gs.getUnits();
int workers = 0;
int base = 0;
int barracks = 0;
for(Unit u : units) {
```

```
    if(u.getPlayer() == player) {
        if(u.getType() == workerType)
            workers++;
        else if(u.getType() == baseType &&\
        gs.getActionAssignment(u) == null)
            base++;
        else if(u.getType() == barracksType)
            barracks++;
    }
}
int workerLimit = (barracks == 0) ? 4 : workers * base;
for(Unit u : units) {
    if(u.getType() == baseType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null && workers < workerLimit &&\
    p.getResources() >= workerType.cost) {
        train(u, workerType);
        break;
    }
}
```

P8:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType() == barracksType && u.getPlayer() == player &&\
    gs.getActionAssignment(u) == null) {
        int nLight = 0;
        int nRanged = 0;
        int nHeavy = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == lightType
                    && u.getPlayer() == p.getID()) {
                nLight++;
            }
            if (u2.getType() == rangedType
                    && u.getPlayer() == p.getID()) {
                nRanged++;
            }
            if (u2.getType() == heavyType
                    && u.getPlayer() == p.getID()) {
                nHeavy++;
            }
        }
        if (nLight == 0 && p.getResources() >= lightType.cost) {
```

```
            train(u, lightType);
        } else if (nRanged == 0 &&\
        p.getResources() >= rangedType.cost) {
            train(u, rangedType);

        } else if (nHeavy == 0 &&\
        p.getResources() >= heavyType.cost) {
            train(u, heavyType);
        }
        if (nLight != 0 && nRanged != 0 && nHeavy != 0) {
            int number = r.nextInt(3);
            switch (number) {
                case 0:
                    if (p.getResources() >= (lightType.cost)) {
                        train(u, lightType);
                    }
                    break;
                case 1:
                    if (p.getResources() >= (rangedType.cost)) {
                        train(u, rangedType);
                    }
                    break;
                case 2:
                    if (p.getResources() >= (heavyType.cost)) {
                        train(u, heavyType);
                    }
                    break;
            }
        }
    }
}
```

Explanation:

This strategy for playing MicroRTS focuses on training units at the barracks owned by the player. The process begins by checking each unit on the game map. If the unit is a barracks owned by the player and it's not currently assigned any action, the strategy proceeds to the next step. The strategy then counts the number of each type of combat units - Light, Ranged, and Heavy - currently owned by the player. Following this, the strategy tries to train one of each type of combat units, starting with the Light unit, then Ranged unit, and finally the Heavy unit. However, this is only done if the player has enough resources to train the unit and doesn't already own a unit of that type. In a situation where the player already owns at least one of

each type of combat units, the strategy randomly chooses between the three types of units to train next. The strategy ensures that the player has enough resources to train the chosen unit type before proceeding with the training. The overall goal of this strategy is to ensure a balanced mix of combat units for the player, while also taking into consideration the amount of resources available to the player.

Best Reconstructed Program for P8:

```
List<Unit> units = gs.getPhysicalGameState().getUnits();
 int numLight = 0, numRanged = 0, numHeavy = 0;
 for(Unit u : units) {
     if(u.getType() == barracksType && u.getPlayer() == player \
     && gs.getActionAssignment(u) == null) {
         for(Unit u2 : units) {
             if(u2.getPlayer() == player) {
                 if(u2.getType() == lightType) {
                     numLight++;
                 } else if(u2.getType() == rangedType) {
                     numRanged++;
                 } else if(u2.getType() == heavyType) {
                     numHeavy++;
                 }
             }
         }
         if(numLight == 0 && p.getResources() >= lightType.cost)
         {
             train(u, lightType);
         }
         else if(numRanged == 0 &&\
         p.getResources() >= rangedType.cost) {
             train(u, rangedType);
         }
         else if(numHeavy == 0 &&\
         p.getResources() >= heavyType.cost) {
             train(u, heavyType);
         }
         else {
             int r = new Random().nextInt(3);
             if(r == 0 && p.getResources() >= lightType.cost) {
                 train(u, lightType);
             } else if(r == 1 && \
             p.getResources() >= rangedType.cost) {
```

```
                    train(u, rangedType);
                } else if(r == 2 && \
                p.getResources() >= heavyType.cost) {
                    train(u, heavyType);
                }
            }
        }
    }
```

P9:

```
for (Unit u : pgs.getUnits()) {
    if (u.getType().canAttack && !u.getType().canHarvest &&\
    u.getPlayer() == player && gs.getActionAssignment(u) == null) {
        Unit closestEnemy = null;
        int closestDistance = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getPlayer() >= 0 && u2.getPlayer() != p.getID())
            {
                int d = Math.abs(u2.getX() - u.getX()) + \
                Math.abs(u2.getY() - u.getY());
                if (closestEnemy == null || d < closestDistance) {
                    closestEnemy = u2;
                    closestDistance = d;
                }
            }
        }
        if (closestEnemy != null) {
            attack(u, closestEnemy);
        }
    }
}
```

Explanation:

In this MicroRTS strategy, the focus is on using your combat units effectively to attack the enemy. The strategy begins by examining all the units on the map. It is particularly interested in the combat units that belong to your team and are not currently assigned any action. A combat unit is identified as a unit that can attack but cannot harvest resources. Upon identifying such a unit, the strategy shifts its focus to finding the closest enemy unit. This is achieved by examining the location of all units on the map that belong to the opposing team. The distance between your combat unit and each enemy unit is calculated using their x and y coordinates on the game map. The enemy

unit with the smallest calculated distance is identified as the closest enemy. Once the closest enemy is identified, your combat unit is instructed to attack it. This sequence is repeated for all your available combat units that are not currently assigned any action. In this way, this strategy ensures that your combat units are always actively engaged in attacking the enemy and focuses on the enemy units that are closest to your units for quick engagement. The overall goal of this strategy is to maintain a high level of offensive pressure on the opponent by continuously engaging the closest enemy units with your available combat units. This approach keeps the opponent on the defensive and could potentially overwhelm them if their defenses are not strong enough.

Best Reconstructed Program for P9:

```
for(Unit u: pgs.getUnits()) {
    if(u.getPlayer() == player && u.getType().canAttack &&\
    !u.getType().canHarvest) {
        if(gs.getActionAssignment(u) == null) {
            Unit closestEnemy = null;
            int closestDistance = Integer.MAX_VALUE;
            for(Unit u2: pgs.getUnits()) {
                if(u2.getPlayer() >= 0 && u2.getPlayer() != player)
                {
                    int dx = u2.getX() - u.getX();
                    int dy = u2.getY() - u.getY();
                    int distance = dx*dx + dy*dy;
                    if(closestEnemy == null || \
                    distance<closestDistance) {
                        closestEnemy = u2;
                        closestDistance = distance;
                    }
                }
            }
            if(closestEnemy != null) {
                attack(u, closestEnemy);
            }
        }
    }
}
```

P10:

```
for(Unit u:pgs.getUnits()) {
    if (u.getType()==baseType && u.getPlayer() == player &&\
```

```
    gs.getActionAssignment(u)==null) {
            if (p.getResources()>=workerType.cost) {
                train(u, workerType);
            }
        }
    }
```

Explanation:

In this strategy, the game goes through each unit on the game map. It looks for any base units that belong to the player and is currently not performing any actions. Once such a base is found, the strategy checks the player's resources. If the player has sufficient resources to train a worker, the base is commanded to train a worker unit. The cost of training a worker is deducted from the player's resources. This strategy primarily focuses on constantly producing workers as long as there's enough resources and the base is idle. The overall goal of this strategy is to maximize the number of worker units the player has, as long as they can afford it.

Best Reconstructed Program for P10:

```
List<Unit> units = pgs.getUnits();
    for(Unit u : units) {
        if (u.getPlayer() == player) {
            if (u.getType() == baseType &&\
            gs.getActionAssignment(u) == null) {
                if (p.getResources() >= workerType.cost) {
                    train(u, workerType);
                }
            }
        }
    }
```